INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Improving the Efficiency of AC Matching and Unification

Steven Mark EKER

N° 2104
Novembre 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Improving the Efficiency of AC Matching and Unification
Amélioration de l'efficacité du filtrage et de l'unification AC

Steven M. Eker

INRIA Lorraine & CRIN
615, rue du Jardin Botanique, BP101, 54600 Villers-les-Nancy
France

November 1993

Improving the Efficiency of AC Matching and Unification

Steven M. Eker

Abstract

This report consists of three independent parts that each study important steps in the matching and unification process for AC theories.

In the first part we consider the problem of AC matching where there is just a single (variadic) AC symbol and no free function symbols in the pattern and subject. We show that even this restricted problem is NP-complete. We give some search methods and empirical results.

In the second part we consider the full AC matching problem where there is no restriction on AC and free functions symbols allowed in the pattern and subject. Our approach is to build a hierarchy of bipartite graph matching problems which encodes all the possible solutions of subproblems. Certain sets of solutions to the graph problems are then used to construct simplified AC systems which are solved by a constrained search.

In the final part we focus on one of the computationally intensive steps in current AC unification algorithms: the extraction of potential unifiers from a Diophantine basis. We show that certain subproblems are NP-complete and derive a new search algorithm which is shown to be at worst equivalent to the best published algorithm and which is potentially much better.

Key Words: Associativity, AC Unification, AC matching, Commutativity, Congruence Class Rewriting, Diophantine Equations, Ordered Normal Form, Poset Searching.

Amélioration de l'efficacité du filtrage et de l'unification AC

Steven M. Eker

Résumé

Ce rapport consiste en trois parties indépendantes dont chacune aborde une étape importante dans les processus de filtrage et d'unification dans les théories associative et commutative (AC).

Dans la première partie, nous considérons le problème du filtrage AC mettant en jeu un seul symbole AC et aucun symbole libre. Nous montrons que même ce cas très restreint est NP-complet. Nous donnons par ailleurs des méthodes de recherche et des résultats obtenus par notre implantation.

Dans la seconde partie, nous considérons le cas général du filtrage AC en présence de symboles libres et de plusieurs symboles AC. Notre approche consiste alors à construire une hiérarchie de problèmes de filtrage de graphes biparties qui encodent toutes les solutions possibles des sous problèmes. Les ensembles complets de solutions aux problèmes de graphe sont alors utilisés pour construire des systèmes simplifiés AC qui sont alors résolus par une recherche contrainte.

Dans la dernière partie, nous nous intéressons à l'un des pas les plus coûteux en calcul dans les algorithmes d'AC-unification actuels : l'extraction des AC-unificateurs potentiels à partir d'une base Diophantienne. Nous montrons que certains sous-problèmes sont NP-complets et nous dérivons un nouvel algorithme de recherche qui est montré être au pire équivalent aux meilleurs algorithmes publiés et qui est potentiellement plus performant.

Mots clefs: Associativité, AC unification, AC matching, Commutativité, Réécriture modulo, Equation Diophanienne, Forme normale ordonnée.

[0]The author's address current address is: Department of Computer Science, City University, Northampton Square, London EC1V 0HB, England.

Contents

Chapter 1

Elementary Associative-Commutative Matching

An elementary associative-commutative (AC) matching problem has a pattern term which consist of a single (variadic) AC function symbol with only variable symbols as arguments and a subject term which consists of a single (variadic) AC function symbol with only constant symbols as arguments. We show that even this very restricted formulation of AC matching has an NP-complete decision problem. We consider a number of methods to contain the growth in the search space including a lookup table for the solubility of subproblems, a digraph reformulation of the problem and a search tree pruning method that uses failure information together with a partial ordering on branches. We give empirical results for the method that seems to work best in practice and list some 'hard' problem instances.

1.1 Introduction

We assume the reader to be familiar with the basic ideas of term rewriting (see for example [6]). The problem of associative-commutative (AC) matching is as follows: Given a pattern term p and a subject term s we wish to find substitutions σ such that $AC \vdash p\sigma = s$ where

$$AC = \{f(X,Y) = f(Y,X), f(f(X,Y),Z) = f(X,f(Y,Z)) \mid f \in \Sigma_{AC}\}$$

is the set of associativity and commutativity axioms for some subset Σ_{AC} of the set Σ of function symbols. Associative-commutative matching has applications in theorem proving [12] and algebraic programming [13, 10]. Various algorithms for AC matching are described in [17, 18, 14, 21]; also AC matching can be seen as a special case of AC unification [24, 23, 20, 4, 3, 8, 1]. AC matching with free function symbols was shown to be NP-complete in [2].

When dealing with function symbols which obey the associativity axiom it is common to *flatten* terms by replacing nested occurrences of such function symbols by variadic function symbols. For example

$$f(a, f(f(g(a,b), g(g(b,c),c)),c))$$

where $f, g \in \Sigma_{AC}$ becomes

$$f^*(a, g^*(a,b), g^*(b,c,c), c).$$

We will consider the problem of associative-commutative matching in 'pure' or 'elementary' case where the (flattened) pattern term consists of a variadic AC function symbol with variable arguments and the (flattened) subject term consists of the same variadic AC function symbol with constant arguments. We will assume that identical arguments are grouped together and write them as a single argument with a superscripted multiplicity. Thus we write a problem instance as

$$f^*(X_1^{\alpha_1}, \ldots, X_n^{\alpha_n}) \leq^?_{AC} f^*(a_1^{\beta_1}, \ldots, a_m^{\beta_m}). \tag{1.1}$$

Such problems typically arise as the last stage of the solution of a full matching problem where any bound variables have been eliminated and the constants are used to abstract ground terms. Since in any matching substitution a variable can be assigned any non-zero number of constants we essentially have to find ways of dividing up the constants $a_1, \ldots, a_m$ with multiplicities $\beta_1, \ldots, \beta_m$ amongst variables $X_1, \ldots, X_n$ with multiplicities $\alpha_1, \ldots, \alpha_n$ such that each variable gets at least one constant.

For complexity purposes we put

$$N = \sum_{i=1}^{n} \alpha_i \quad \text{and} \quad M = \sum_{j=1}^{m} \beta_j.$$

Intuitively N is the total number of variable occurrences and M is the total number of constant occurrences.

If we denote the number of a_j's assigned to variable X_i by $x_{i,j}$ then clearly the solutions of (1.1) correspond to the set of non-negative solutions to the system of independent linear inhomogeneous Diophantine equations

$$\begin{array}{ccccccc} \alpha_1 x_{1,1} & + & \ldots & + & \alpha_n x_{n,1} & = & \beta_1 \\ \vdots & & & & \vdots & & \vdots \\ \alpha_1 x_{1,m} & + & \ldots & + & \alpha_n x_{n,m} & = & \beta_m \end{array}$$

such that for each $i \in \{1, \ldots, n\}$

$$\sum_{j=1}^{m} x_{i,j} \geq 1. \tag{1.2}$$

Theorem 1 *The problem of deciding whether a given elementary AC matching problem has a solution is NP-complete.*

Proof: It is trivial to see that the decision problem is in NP. To show that it is NP-hard we show that the equivalent Diophantine equation problem described above is *NP-complete in the strong sense*; i.e. that it remains NP-complete even if numbers occurring in problem instances are bounded by a polynomial over number of symbols needed to represent such instances under a 'reasonable' representation scheme. We do this by reducing the following problem (proved NP-complete in the strong sense in [11]) to it.

3-PARTITION: Given a finite set $A = \{a_1, \ldots, a_{3m}\}$ of elements, a bound $B \in \mathbf{Z}^+$ and a size $s(a) \in \mathbf{Z}^+$ for $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$, decide whether A can be partitioned into m disjoint sets $S_1, \ldots, S_m$ such that for $j \in \{1, \ldots, m\}$, $\sum_{a \in S_j} s(a) = B$.

We form an instance of the Diophantine equation problem

$$\begin{array}{ccccccc} s(a_1) x_{1,1} & + & \ldots & + & s(a_{3m}) x_{3m,1} & = & B \\ \vdots & & & & \vdots & & \vdots \\ s(a_1) x_{1,m} & + & \ldots & + & s(a_{3m}) x_{3m,m} & = & B \end{array}$$

with the condition that for $i \in \{1, \ldots, n\}$

$$\sum_{j=1}^{m} x_{i,j} \geq 1.$$

If this instance has a solution then we have $\sum_{j=1}^{m} x_{i,j} = 1$ (since otherwise the sum of the left hand sides of the equations would be greater that the sum of the right hand sides) and thus we have a solution to the instance of 3-PARTITION by putting

$$S_j = \{a_i \mid x_{i,j} = 1\}.$$

Also if the instance of 3-PARTITION has a solution then we get a solution of our Diophantine equation problem instance by putting

$$x_{i,j} = \begin{cases} 0 & \text{if } a_i \notin S_j \\ 1 & \text{if } a_i \in S_j \end{cases}$$

Also the instance of the Diophantine equation problem clearly corresponds to an elementary AC matching problem. □

Remark 1 *It is not sufficient to use an NP-complete problem that is not NP-complete in the strong sense in the above proof since we measure the size of our matching problems by N and M. Effectively we are representing the corresponding Diophantine equation problem in unary which is not a 'reasonable' representation. A pseudo-polynomial time algorithm for deciding the corresponding Diophantine equation problem would immediately gives us a polynomial time algorithm for deciding the original matching problem.*

Remark 2 *Each Diophantine equation on its own can be solved in pseudo-polynomial time by dynamic programming. Thus it is the condition (1.2) on the acceptable combination of solutions that causes the whole Diophantine equation problem to be NP-complete in the strong sense and the original matching problem to be NP-complete. In elementary associative-commutative-identity (ACU) matching this condition does not apply since (formal) variables that are not assigned a (formal) constant can be assigned an identity element. Thus elementary ACU matching can be decided in polynomial time.*

1.2 Search methods

Although the linear inhomogeneous Diophantine equations are independent (they have no shared variables) because we have condition (1.2) solving them independently and combining the solutions is a very inefficient way to solve the problem. If the jth equation had S_j solutions we would have to consider $\prod_{j=1}^{m} S_j$ combinations of solutions many of which would fail condition (1.2).

Instead we solve the equations in parallel, at the i step considering only the variables $x_{i,1}, \ldots, x_{i,m}$. To reduce the branching we will assume $\alpha_1, \ldots, \alpha_n$ are sorted in descending order and tackle the variables with greatest multiplicity first. To avoid branches that must fail we enforce constraints at each step.

1.2.1 Multiplicity of each constant

At step i, we define

$$\beta_{i,j} = \beta_j - \sum_{k=1}^{i-1} \alpha_k x_{k,j}.$$

Intuitively $\beta_{i,j}$ denotes the number of a_j's available to be assigned at step i. Clearly for each j we must choose $x_{i,j}$ such that

$$x_{i,j} \leq \frac{\beta_{i,j}}{\alpha_i}$$

since we can not allocate more a_j to x_i than the number we have left at this stage, divided by the multiplicity of x_i.

1.2.2 Total multiplicities of remaining variables and constants

At step i we must choose $x_{i,1}, \ldots, x_{i,m}$ such that

$$\sum_{j=1}^{m} x_{i,j} \leq \frac{\sum_{j=1}^{m} \beta_{i,j} - \sum_{k=i+1}^{n} \alpha_k}{\alpha_i}$$

since we must ensure there is at least one unassigned constant left for each remaining variable.

1.2.3 Solubility constraints

After step i we may view the remaining problem as a smaller version of the original problem in which the variables $x_{1,j}, \ldots, x_{i,j}$ for each j have been eliminated and the constant terms $\beta_1, \ldots, \beta_m$ have been reduced to $\beta_{i,1}, \ldots, \beta_{i,m}$. Clearly if one of the equations

$$\alpha_{i+1} x_{i+1,j} + \ldots + \alpha_n x_{n,j} = \beta_{i,j}$$

does not have a solution then this branch of the search tree will fail. For $b \in \mathbf{N}$ and $\underline{v} \in \mathbf{N}^k$ we define the relation

$$split(b, \underline{v}) \Leftrightarrow (\exists \underline{u} \in \mathbf{N}^k).[\underline{u}.\underline{v} = b]$$

where $\underline{u}.\underline{v}$ is the scalar product of the two vectors. Clearly the following properties hold:

1. $split(0, \underline{v})$
2. $split(b, \underline{v}) \wedge split(c, \underline{v}) \Rightarrow split(b + c, \underline{v})$
3. $split(b, \underline{v}) \wedge \lambda \in \mathbf{N} \Rightarrow split(\lambda b, \underline{v})$
4. $split(v_{k+1}, (v_1, \ldots, v_k)) \Rightarrow (split(b, (v_1, \ldots, v_k)) \Leftrightarrow split(b, (v_1, \ldots, v_{k+1})))$

Now at step i, for each $j \in \{1, \ldots, m\}$ we must choose $x_{i,j}$ such that

$$split((\beta_{i,j} - \alpha_i x_{i,j}), (\alpha_{i+1}, \ldots, \alpha_n))$$

otherwise the number of b_j's remaining cannot be divided up among $x_{i+1}, \ldots, x_n$. Clearly if $\alpha_n = 1$ this condition is always true. Otherwise we can enforce it by keeping a boolean vector of length $\beta_{\max} = \max_{j=1}^{m} \beta_j$ for every distinct α_i, $i \in \{2, \ldots, n\}$ where the kth entry of the vector associated with α_i is true iff $split(k, (\alpha_i, \ldots, \alpha_n))$. Clearly if α_i and α_{i+1} are not distinct, that is $\alpha_i = \alpha_{i+1}$, then they can share the same vector. Thus the storage requirement is maximised when all the α_i's are distinct. Since $\alpha_n > 1$ the worst case occurs when $\alpha_i = 2 + n - i$ for $i \in \{2, \ldots, n\}$ and $\alpha_1 = n$. Then

$$N = \frac{n^2 + 3n - 2}{2}$$

Thus

$$n \leq \left(2N + \frac{17}{4}\right)^{\frac{1}{2}} - \frac{3}{2}$$

and hence total storage in bits for the boolean vectors is bounded by

$$\left(\left(2N + \frac{17}{4}\right)^{\frac{1}{2}} - \frac{5}{2}\right)\beta_{\max}$$

giving a $O(M\sqrt{N})$ space complexity bound. To initialize the vector associated with α_n we need $O(\beta_{\max})$ operations. The kth component of the boolean vector associated with α_i can be found by testing the components $k, k+\alpha_i, k+2\alpha_i, \ldots$ of the (previously calculated) vector associated with α_{i+1} until we find a true element or run out of components. Thus to initialize the vector associated with α_i we need to make at most

$$\sum_{k=1}^{\beta_{\max}} \left\lfloor 1 + \frac{\beta_{max} - k}{\alpha_i} \right\rfloor$$

tests. This is bounded by

$$\beta_{\max} + \frac{\beta_{\max}(\beta_{\max} - 1)}{2\alpha_i}$$

The worst case is the same as the one above for storage and we have, summing over the vectors associated with $\alpha_2, \ldots, \alpha_{n-1}$ a bound of

$$\sum_{k=3}^{n} \left(\beta_{\max} + \frac{\beta_{\max}(\beta_{\max} - 1)}{2k}\right) = (n-2)\beta_{\max} + \frac{\beta_{\max}(\beta_{\max} - 1)}{2}\left(H_n - \frac{3}{2}\right)$$

where H_n is the nth harmonic number. Thus we have a $O(M\sqrt{N} + M^2 \log N)$ time complexity bound for initializing this data structure. For implementation purposes it is better to use vectors of length $\beta_{\max} + 1$ to avoid treating zero as a special case.

```
//      We want to select r elements from a multiset {e_1^{k_1}, ..., e_m^{k_m}}.
p := m;
loop:
    while p > 1 and Σ_{i=1}^{p-1} k_i ≥ r do
        s_p := 0; p := p − 1
    od
    s_p = r − Σ_{i=1}^{p-1} k_i;
    for p := p − 1 downto 1 do
        s_p := k_p
    od;
//      use selection {e_1^{s_1}, ..., e_m^{s_m}}
    r := s_1;
    for p := 2 to m do
        if r > 0 and s_p < k_p then
            s_p := s_p + 1; r := r − 1; p := p − 1;
            goto loop
        else
            r := r + s_p
        fi
    od
```

Figure 1.1: Iterative algorithm for enumerating selections from a multiset

1.2.4 First algorithm

The first algorithm consists of a recursive procedure which finds substitutions for x_i and on each such substitution calls itself to find substitutions for the remaining variables. We keep track of the 'unused multiplicity' $\beta_{i,j}$ of each constant a_j for $j \in \{1, \ldots, m\}$ in a global array which is incrementally updated when the value of some $x_{i,j}$ is changed. Thus at the start of each invocation of the procedure for x_i we know that $x_{i,j}$ must lie in $\{0, \ldots, \beta_{i,j}/\alpha_i\}$. If $\alpha_n > 1$ we can narrow this down further by the constraint in §1.2.3. Also we know that $\sum_{j=1}^{j=m} x_{i,j}$ must be at least 1 by condition (1.1) and no more than

$$\sum_{j=1}^{m} \beta_{i,j} - \sum_{k=i+1}^{n} \alpha_k$$

by the constraint in §1.2.2. Furthermore both these lower and upper bounds might be improved by the bounds we have on individual $x_{i,j}$'s. Any such lower bounds can be assigned immediately and a recursive call made to try this assignment. It is also possible that the improved lower bound could exceed our original upper bound, leading to immediate failure of this branch. Then suppose the best upper bound on $\sum_{j=1}^{j=m} x_{i,j}$ is q. For each $r \in \{1, \ldots, q\}$ we search for ways of assigning r extra constants to x_i respecting upper bounds and solubility constraints on each $x_{i,j}$. This is really the problem of choosing all combinations of r elements from a multiset (possibly with extra restrictions) and can be handled by an iterative algorithm such as that given in Figure 1.1.

Consider the following problem:

$$f^*(X_1^{65}, X_2^{63}, X_3^{61}, X_4^{59}, X_5^{57}, X_6^{55}, X_7^{53}, X_8^{51}, X_9^{49}, X_{10}^{48}, X_{11}^{47}, X_{12}^{46}, X_{13}^{45}, X_{14}^{44}, X_{15}^{43}, X_{16}^{42}) \leq^?_{AC} f^*(a_1^{100}, a_2^{101}, a_3^{102}, a_4^{103}, a_5^{104}, a_6^{105}, a_7^{106}, a_8^{107})$$

The above algorithm finds the single substitution very quickly. Now if we change the problem slightly by including x_{17}^1 as a left hand side argument and a_9^1 as a right hand side argument the running time increases by more than four orders of magnitude. The reason for the change is that with $\alpha_{17} = 1$, the solubility constraints no longer apply and the search tree has many more branches.

1.2.5 Subtotal constraints

Consider the problem at step i before any choices have been made. Now for each $h \in \{i+1, \ldots, n\}$ only those (formal) constants a_j with $\beta_{i,j} \geq \alpha_h$ may be assigned to (formal) variables $x_i, \ldots, x_h$. Thus for a given branch not to fail it is necessary that for each such h,

$$\sum_{k=i}^{h} \alpha_k \leq \sum_{\beta_{i,j} \geq \alpha_h} \beta_{i,j}$$

These constraints are not so easy to integrate into the main multiset enumeration loop for x_{i-1} instead we do a brute force test of these constraints at the beginning of the procedure for x_i and terminate the branch as soon as one of them fails. With this naive approach the run time for the original example increases slightly while the run time for the modified example drops to approximately that for the original example.

1.2.6 A tighter subtotal constraint

The constraints of §1.2.5 can be tightened as follows: We define $maxuse_{\gamma,\delta}(\beta)$ to be the maximum of $\sum_{k=\gamma}^{\delta} s_k$ taken over all non-negative solutions $s_\gamma, \ldots, s_n$ of the Diophantine equation

$$\alpha_\gamma s_\gamma + \cdots + \alpha_n s_n = \beta$$

Intuitively *maxuse* is the maximum number of assignments of some constant of (unused) multiplicity β that can be made to variables $x_\gamma, \ldots, x_\delta$ taking solubility constraints into account. Now for a given branch at step i not to fail it is necessary that for each $h \in \{i+1, \ldots, n\}$ that

$$\sum_{k=i}^{h} \alpha_k \leq \sum_{j=1}^{m} maxuse_{i,h}(\beta_{i,j})$$

It is not clear whether this tighter constraint can be enforced efficiently.

1.2.7 Making use of repeated multiplicities

Adjacent variables $x_i, \ldots, x_k$ such that $\alpha_i = \cdots = \alpha_k$ can be treated as single variables to which at least $k - i + 1$ constants must be assigned. Constraints on the original search space can be adapted to the reduced search space and the solutions for the original problem corresponding to a solution of the reduced problem can be recovered by multiset partitioning.

A similar technique can be applied to the constants. Here we are interested in $\beta_{i,j}$, that is the unused multiplicity of a constant. Instead of keeping track of the unused multiplicity of a particular constant we keep track of the number of constants with a given unused multiplicity and do not distinguish between them. There is then the possibility of reducing the search space even if initially the multiplicity of each constant is distinct.

This leads to a formulation of elementary AC matching as the search for paths in a $(k+1)$-partite directed acyclic graph where k is the number of distinct variable multiplicities. Here a node in the 1st part is labelled with an initial constant multiplicity while a node in the $(i+1)$th part is labelled with a potential 'left over' constant multiplicity after assignments to all variables with the ith greatest distinct multiplicities. A node labelled γ in the ith part has arcs to nodes in the $(i+1)$th with label δ such $\gamma - \delta$ is divisible by the ith distinct variable multiplicity and such an arc is labelled with the result of this division. A potential solution is a multiset of paths, one path for each constant such that the path for a constant of multiplicity γ starts at the node labelled γ in the 1st part and ending at the single node labelled 0 in the $(k+1)$th part. A potential solution is really a solution if the sum of the labels on all selected edges from the ith part into the $(i+1)$th part is greater than or equal to the number of variables having the ith greatest distinct multiplicity. For example the graph representation of the problem

$$f^*(X_1^9, X_2^7, X_3^5, X_4^5, X_5^1) \leq_{AC}^{?} f^*(a_1^{20}, a_2^6, a_3^6)$$

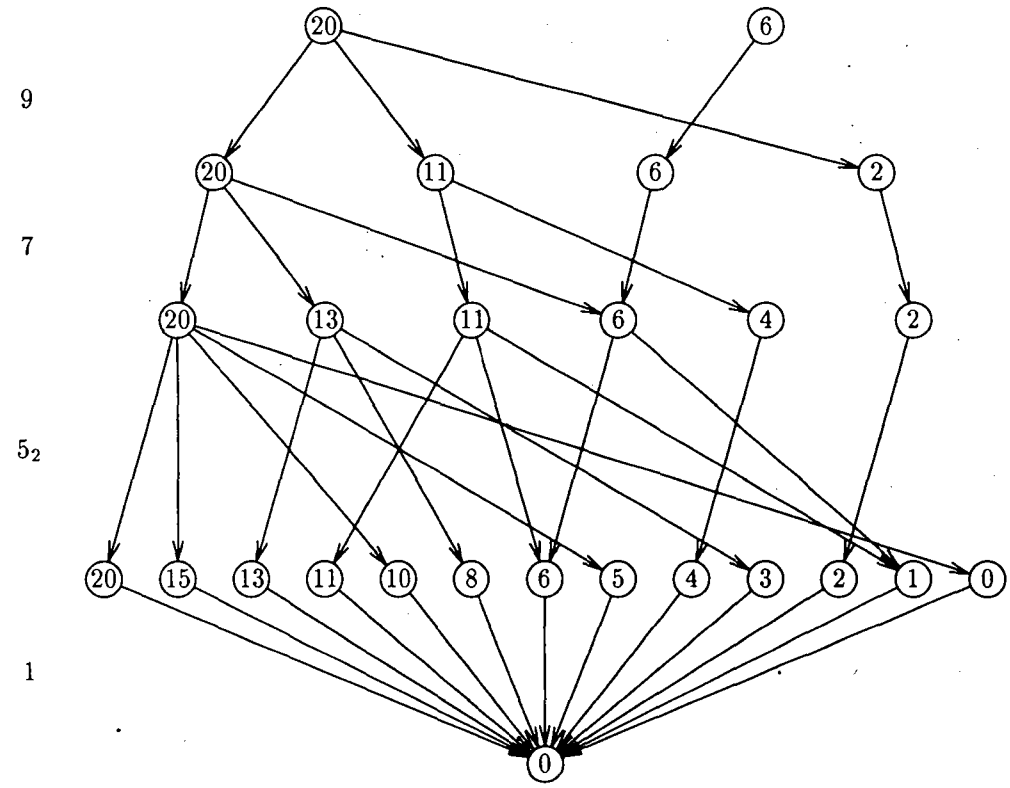

Figure 1.2: Graph representation of an elementary AC matching problem

is shown in Figure 1.2 with edge labels omitted for clarity. Both solutions to the problem are represented by the multiset of paths {20-11-4-4-0, 6-6-6-1-0, 6-6-6-1-0}. Since at each stage constants with the same remaining multiplicity are interchangable, we are really only interested in the set of selected edges rather than the set of paths (since several sets of paths may give rise to the same set of edges). However transforming the problem in this way makes enumerating the possibilities for each solution step much more complicated and recovering solutions to the original problem from a solution of the reduced problem is also complicated. The idea of combining repeated coefficients to speed up the solution of Diophantine equations was proposed in [5] where a somewhat different digraph reformulation of a related Diophantine problem is given.

1.2.8 Using failure information

Whenever we make some choice for $x_{i,1}, \ldots, x_{i,m}$, say $\gamma_1, \ldots, \gamma_m$, we will either succeed or fail to find a solution with these assignments. If we fail we may be able to use the information that was no solution with these assignments to prune the search space for future assignments. In particular we can rule out any assignment $\delta_1, \ldots, \delta_m$ such that for $j \in \{1, \ldots, m\}$,

$$\delta_j \geq \gamma_j \wedge \mathit{split}(\alpha_i(\delta_j - \gamma_j), (\alpha_{i+1}, \ldots, \alpha_n))$$

The reason being that if we could get a solution with $x_{i,j} = \delta_j$ for $j \in \{1, \ldots, m\}$ then we could modify this solution by putting $x_{i,j} = \gamma$ and distributing the remaining $\alpha_i(\delta_j - \gamma_j)$ over $x_{i+1,j}, \ldots, x_{n,j}$ contradicting the fact that there was no solution with $x_{i,j} = \gamma_j$ for $j \in \{1, \ldots, m\}$.

To use this failure information systematically at step i we define ordering

$$\gamma \sqsubseteq_i \gamma' \Leftrightarrow \gamma \leq \gamma' \wedge \mathit{split}(\alpha_i(\gamma' - \gamma), (\alpha_{i+1}, \ldots, \alpha_n))$$

Lemma 1 *$\sqsubseteq_i$ is a partial order.*

Proof: Reflexivity and antisymmetry are obvious. For transitivity, suppose $\gamma \sqsubseteq_i \gamma'$ and $\gamma' \sqsubseteq_i \gamma''$. Clearly $\gamma \leq \gamma''$. Also

$$split(\alpha_i.(\gamma'' - \gamma), (\alpha_{i+1}, \ldots, \alpha_n)) = split(\alpha_i.(\gamma'' - \gamma') + \alpha_i.(\gamma' - \gamma), (\alpha_{i+1}, \ldots, \alpha_n))$$

which must be true since $split(\alpha_i.(\gamma'' - \gamma'), (\alpha_{i+1}, \ldots, \alpha_n))$ and $split(\alpha_i.(\gamma' - \gamma), (\alpha_{i+1}, \ldots, \alpha_n))$. □

Now for each $x_{i,j}$ we will have some set P_j of assignments that meet the other constraints. We extend the partial order $\sqsubseteq_i$ to m-tuples by

$$(\gamma_1, \ldots, \gamma_m) \sqsubseteq_i (\gamma'_1, \ldots, \gamma'_m) \Leftrightarrow \forall j \in \{1, \ldots, m\}.[\gamma_j \sqsubseteq_i \gamma'_j]$$

Now each potential assignment $\gamma_1, \ldots, \gamma_m$ is only tried if all its predecessors under this ordering resulted in at least one solution. In practice we only need to test the immediate predecessors $\gamma_1, \ldots, \gamma_m$.

In order to implement this technique, at step i we generate and store each poset $\langle P_j, \sqsubseteq_i \rangle$ of potential assignments to $x_{i,j}$ as a transitively-reduced directed acyclic graph where each element of P_j is represented by a node with a linked list of pointers to the nodes representing its immediate predecessors. Since the partial order $\sqsubseteq_i$ is trivially embedded in the total ordering $\leq$ we generate the members of P_j in ascending order and each member can be inserted into the transitively-reduced directed acyclic graph in linear time by traversing the existing nodes in reverse order and whenever an immediate predecessor is found, marking all of its predecessors.

Once we have computed each poset $\langle P_j, \sqsubseteq_i \rangle$ for $j \in \{1, \ldots, m\}$ we need to generate the m-tuples in ascending order and keep track of those for which at least one solution was subsequently found. Generating the m-tuples is done by embedding the partial order on m-tuples in the total lexicographic order on tuples of integers. Then we generate m-tuples by a straightforward lexicographic search on the totally ordered sets $\langle P_j, \leq \rangle$. For space and time efficiency the set of m-tuples for which at least one solution was subsequently found is constructed as a digital search tree where each root to leaf path is of length m.

1.3 Experimental results

The algorithm which we give detailed results for is based on the constraints up to and including that of §1.2.5. One extra improvement implemented is that the existance of a variable with multiplicity 1 is checked for and in this case solubility information is not computed and a simplified search routine is called. The algorithm was run on problems with $n = m = 5, \ldots, 20$. For each size 10000 problems were randomly generated with $\alpha_i \in \{1, \ldots, 40\}$ and $\beta_j \in \{5, \ldots, 50\}$ and the first solution (if one existed) was sought. For each size the number of failures and successes, the average and worst time for failures and successes and the actual worst case examples were recorded. The results (made on a SPARC IPC) are shown in table 1.1; timings are in milliseconds with 10ms granularity. As the problems get larger, the average time taken to solve a problem increases rapidly but a large proportion of this time is taken up by a few 'hard' instances, and the maximum time taken by any one problem increases dramatically. The hard instances that have solutions are listed in table 1.2 and those that have no solution are listed in table 1.3. Notice that problems which have no solution generally consume more time than those which have solutions because for the former we must exhaustively explore a (potentially) large search tree whereas for the latter we stop once we have found one solution.

There did not seem to be a machine efficient way of enforcing the constraint of §1.2.6 while the graph based optimisation of §1.2.7 would have required complicated enumeration algorithms. The search tree pruning method of §1.2.8 was also tried but it was found that the overhead of maintaining and searching the extra data structures outweighed the savings due to a smaller search space although the difference became smaller on harder problems and it is possible that this method may be faster on even harder problems.

1.4 Concluding remarks

We have shown that the problem of elementary AC matching is NP-complete and have presented a number of improvements over the naive brute force search. While these techniques cut down the size

n	successes	success time		failure time	
		average	max	average	max
5	1243	0	10	0	10
6	1250	0	10	0	10
7	1340	0	10	0	10
8	1888	0	10	0	10
9	1661	1	10	0	30
10	1697	1	10	0	40
11	1810	1	10	0	100
12	1906	1	40	0	460
13	2027	2	480	1	1510
14	2075	2	150	2	4410
15	2171	6	4670	3	7880
16	2139	3	1000	10	25930
17	2246	5	2120	59	166660
18	2416	19	19010	168	720900
19	2472	21	12720	280	1204310
20	2675	116	68370	195	448730

Table 1.1: Results for random elementary AC problems

n	variable multiplicities	constant multiplicities
13	38,32,27,26,25,23_2,20,19_2,10,5,2	41,40,35_2,33_2,32,30,29,18,7,6,2
14	38,35,31,25,24,22,21,20,18_2,11_2,7,2	43,42,39,35,33,32,27,26,25,16,10,9_2,2
15	38,37_2,32,31,29,28_2,27,24,20,18,17,2,1	44,43_3,40,38,35_2,31,30,26,11_2,6,5
16	36,32,30,29,28_2,24,17,15,13,12,11_3,2,1	45,36_2,34,31,30,24,20,17_2,16,14_2,10,2,1
17	37,33,32,30,27_2,25,23,18_3,16,14,11,7,4,2	44,42_2,39,36,28,27_3,25,24,21,8,7,6,4_2
18	39,33,32,27,24,23,22_2,21,17,16,15_2,12_2,10,3,1	45,39,38,35_2,29,28,25,21_3,19,16,7,4,3_2,1
19	40,39,38,37,36,33,31,29,27,26_2,22_2,9_2,7,6,5,4	45_2,44,43,40,38,37_2,36,35,30,27,25,23,20,16_2,11,9
20	40,36,34,33,32,27_2,26_3,24,23,22_2,15,12,11_2,5,1	45_2,42_2,39,36,35,34,33_2,30,25_2,18,12_2,11,5,3,1

Table 1.2: Hard success instances

n	variable multiplicities	constant multiplicities
13	34,29_2,28,25_2,24_2,22,15,14,3,2	45,43,38,37,34,33,31,28,15,11,7,6,5
14	39,28,27_3,24,23,22,21,20,18,10,8,1	42,37,36_2,35,33,32,29,28,19,16,6,4_2
15	33_2,32,31,28_2,27,24_2,23,19,14,10,7,1	43,41,39,37_2,35_2,29,28,19,18,16,14,6,3
16	38,32_2,31,29,28,27,26,24,23,20,17,13,8,7,1	45,42,39,38_2,37,36,35,34,22,21,20,19,12,10,3
17	37,33_2,32_2,26,24,23_4,19,14,12,6,4,3	43_2,40_2,39,37,36_2,28,24_3,16,15,9_2,3
18	40,39,38_2,33,28,27,26_2,25,24_2,21,15,8,5,4,1	44_2,42_3,40,38,37,29,28,27_2,19,14,13,5,4,2
19	39,37_2,35,33,32,31,30,27,26_3,25,22,12,5,3,2,1	45,44_2,41,40,38,37_2,34_2,32,27,24,18,17,13,6,5,3
20	39,36,34,33,30,29_3,25,23,22_3,16,10,7,4,3_2,1	43,42_2,36,35_2,34,33,31_2,30,28,21,16_4,9,8,3

Table 1.3: Hard failure instances

of the search space they also have an overhead and for problems with a tractable search space, using failure information seems to slow down the search. We have also given a digraph reformulation of the problem but it remains to be seen if this can be searched more efficiently. We have listed some 'hard' instances for our method.

Chapter 2

Associative-Commutative Matching Via Bipartite Graph Matching

We consider the problem of term matching where some subset of the function symbols are associative-commutative. Our approach is to build a hierarchy of bipartite graph matching problems which encodes all the possible solutions of subproblems. Sets of solutions to the graph matching problems which are consistent on variable assignments give rise to what we refer to as semi-pure AC systems in which assignments for every variable occurring under a free function symbol in the pattern are known. These semi-pure AC systems are then solved by an exhaustive search to find complete matching substitutions. We give a number of refinements which considerably cut down the search space at all stages in the algorithm leading to efficient solution of non-pathological problem instances.

2.1 Introduction

Benanav *et al* [2] showed that AC matching with free function symbols is NP-complete, but that if the pattern term was restricted to being linear (i.e. no variable occurs more than once) then a single solution (if one exists) could be found in polynomial time by transforming the problem into a graph matching problem. We extend this idea to finding all matching substitutions with non-linear patterns and incorporate many efficiency improvements although in the worst case pathological non-linear patterns constructed using the method of the NP-completeness proof in [2] could give rise to exponential running times even if there was no matching substitution. Also problem instances with linear patterns may have an exponential number of solutions.

We will use $F, G, H, \ldots$ for AC function symbols, $f, g, h, \ldots$ for free function symbols, $a, b, c, \ldots$ for constant symbols and $L, M, N, \ldots$ for variable symbols. W.l.o.g. we will assume that the subject s contains no variables (we could replace any such variables with new constants). We write down a matching problem as an inequality $p \leq^?_{AC} s$. We call such a problem *semi-pure* if p consists of a single AC function symbol with only variable symbol arguments. If in addition s consists of a single AC function symbol with only constant symbol arguments we call the problem *pure*. We define the size $|t|$ of a term t to be the number of function, constant and variable symbols that it contains.

2.2 Ordered normal form

A term t which contains AC function symbols is a representative of an equivalence class of terms which are equal modulo associativity-commutativity. The ordered normal form [18, 14] gives us a canonical representative for each such equivalence class, so by converting terms to ordered normal form we can check for equality modulo AC by checking for syntactic equality.

It is well known that terms involving associative function symbols can be converted to a normal form by grouping associative operators to the right (or left). Equivalently, and more usefully for machine representation we may flatten such terms by replacing nested occurrences of the same associative operator

by a single variadic operator. The same may be done with AC operators however this does not give a unique normal form modulo associativity-commutativity since the commutativity axiom may be used to arbitrarily permute the arguments of a variadic operator. We obtain a unique normal form by regarding the arguments of a variadic operator as a multiset of terms (since the same term may occur as an argument more than once). Ordered normal form can be seen as an implementation of this idea where a unique syntactic representation of this multiset of arguments is obtained by sorting and grouping.

We suppose there is a total ordering $\geq$ on symbols. This ordering can be inductively extended to terms in ordered normal form. Note that there is a mutual recursion between the definition of the ordering and the definition of ordered normal form since ordered normal form is defined in terms of the ordering. When we have two terms in ordered normal form whose top symbols differ, then the ordering on the terms is given by the ordering on their top symbols. Suppose we have two terms with the same free function symbols as top symbol. Then

$$f(t_1, \ldots, t_n) \geq f(u_1, \ldots, u_n)$$

iff there exists $k \in \{1, \ldots, \min(n, m)\}$ such that $t_j = u_j$ $j \in \{1, \ldots, k-1\}$ and $t_k \geq u_k$ (this is the usual lexicographic ordering applied argument lists). Suppose we have two terms with the same AC top symbol and whose arguments have been sorted in descending order with like terms grouped. We use the notation t^α to denote α copies of a term t and refer to α as the *multiplicity* of t. Then

$$F(t_1^{\alpha_1}, \ldots, t_n^{\alpha_n}) \geq F(u_1^{\beta_1}, \ldots, t_m^{\beta_m})$$

iff there exists $k \in \{1, \ldots, \min(n, m)\}$ such $t_j = u_j$ and $\alpha_j = \beta_j$ for $j \in \{1, \ldots, k-1\}$ and either $t_k > u_k$ or $t_k = u_k$ and $\alpha_k > \beta_k$ (this is the usual multiset ordering applied to multisets of arguments).

By flattening nested AC function symbols and then sorting and grouping the arguments of the (remaining) AC function symbols we arrive at ordered normal form. In our examples we use the total ordering $a \geq \cdots \geq z \geq A \geq \cdots \geq Z$. For example the ordered normal form of

$$f(F(F(N, F(P, g(a, L))), F(N, g(M, b))), G(G(G(U, a), G(h(Q), h(S))), G(G(g(T, a), N), U)), V)$$

is

$$f(F(g(a, L), g(M, b), N^2, P), G(a, g(T, a), h(Q), h(S), N, U^2), V).$$

Flattening a term and combining identical subterms may produce a new term with fewer symbols. However it is easy to show that flattening will never reduce the number of symbols by more than a factor of two and in the worst case there will be no identical subterms that can be combined. Thus for worst case complexity arguments we can safely ignore this reduction in the number of symbols.

The naive way of computing the ordered normal form of a term t is to first flatten it and then sort and group its arguments. Flattening can trivially be done in $O(|t|)$ operations by representing the arguments of each function symbol as a linked list, allowing constant time concatenation. We show sorting of a flattened term into ordering normal form can be done in $O(|t| \log^2 |t|)$ operations using merge sort on linked lists; this is not as completely trivial as it seems since while merge sorting an argument list of n terms requires $O(n \log n)$ comparisons on terms, the number of operations required to compare two terms is not a constant.

We model sorting with non-uniform comparisons costs as follows. Let $E = \{e_1, \ldots, e_n\}$ be a set of elements where each element e has a weight $w(e)$. The cost of comparing two elements e_1, e_2 is $O(\min(w(e_1), w(e_2)))$. This accurately models the cost of comparing terms since in the worst case we need only compare as many symbols as exist in the smaller of two terms.

Theorem 2 *Two sorted sequences s, s' of elements can be merged in*

$$O\left(\sum_{e \in s \cup s'} w(e) - \max_{e \in s \cup s'} w(e)\right)$$

operations.

Proof: We only need count the cost of comparisons since any overhead operations will have cost linear in sum of the lengths of s and s'. Let the cost of the comparisons needed to do the merge be denoted by $c(s, s')$ and let the cost of comparing elements e_1, e_2 be bounded by $K.\min(w(e_1), w(e_2))$. We prove that

$$c(s, s') \leq K\left(\sum_{e \in sUs'} w(e) - \max_{e \in sUs'} w(e)\right)$$

by induction. In the basis case, one of s and s' is empty and the hypothesis trivially holds. In the induction step, the first elements on each sequence are compared, and one of these elements, α is removed and placed on the merged sequence. By the induction hypothesis, the cost of comparisons needed to solve the reduced problem is bounded by

$$K\left(\sum_{e \in (sUs')-\{\alpha\}} w(e) - \max_{e \in (sUs')-\{\alpha\}} w(e)\right).$$

We consider two cases.

Case 1: $w(\alpha) < \max_{e \in sUs'} w(e)$
The cost for the comparison that placed α on the merged sequence is bounded by $w(\alpha)$ and thus $c(s, s')$ is bounded by

$$K\left(w(\alpha) + \sum_{e \in (sUs')-\{\alpha\}} w(e) - \max_{e \in (sUs')-\{\alpha\}} w(e)\right) = K\left(\sum_{e \in sUs'} w(e) - \max_{e \in sUs'} w(e)\right)$$

since $w(\alpha) < \max_{e \in sUs'} w(e)$.

Case 2: $w(\alpha) = \max_{e \in sUs'} w(e)$
Let the element that α was compared against be β. Now the cost of the comparison was $K.w(\beta)$ and $\max_{e \in (sUs')-\{\alpha\}} w(e)$ is bounded from below by $w(\beta)$. So $c(s, s')$ is bounded by

$$\begin{aligned} K\left(w(\beta) + \sum_{e \in (sUs')-\{\alpha\}} w(e) - w(\beta)\right) &= K\left(\sum_{e \in sUs'} w(e) - w(\alpha)\right) \\ &= K\left(\sum_{e \in sUs'} w(e) - \max_{e \in sUs'} w(e)\right) \end{aligned}$$

□

Theorem 3 *Merge sort on E requires at most*

$$O\left(\left(\sum_{i=1}^{n} w(e_i) - \max_{i=1}^{n} w(e_i)\right) \log n\right)$$

operations.

Proof: Clearly merge sort requires $\lceil \log n \rceil$ 'rounds' where the jth round consists of at most $n/2^j$ merges of pairs of sorted sequences whose lengths are bounded by 2^{j-1}. We need to show that each round is accomplished in

$$O\left(\sum_{i=1}^{n} w(e_i) - \max_{i=1}^{n} w(e_i)\right)$$

operations. Since each element occurs in exactly one sequence this follows from Theorem 2. □

Theorem 4 *Converting the flattened version of a term t into ordered normal form using merge sort can be done in $O(|t| \log^2 |t|)$ time.*

Proof: Let the cost of converting a flattened term t into ordered normal form be $cost(t)$. We find expressions to bound this cost in each of four cases.

Case 1: t is a constant symbol.
Then $cost(t) = K_1$ for some constant K_1.

Case 2: t is a variable symbol.
Then $cost(t) = K_2$ for some constant K_2.

Case 3: $t = f(t_1, \ldots, t_n)$.
Then

$$cost(t) \leq K_3 n + \sum_{i=1}^{n} cost(t_i).$$

Case 4: $t = F^*(t_1, \ldots, t_n)$.
Then we have to account for the cost of converting the subterms, the cost of sorting the subterms and the cost of combining identical subterms. The last two costs can be combined by choosing a large enough constant K_4 giving

$$cost(t) \leq K_4 \left(\sum_{i=1}^{n} |t_i| - \max_{i=1}^{n} |t_i| \right) + \sum_{i=1}^{n} cost(t_i)$$

via Theorem 3. We now prove an explicit bound on $cost(t)$ by induction on the structure of t. Our induction hypothesis is that $cost(t) \leq (K \log m)|t| \log |t|$

where $K = \max_{j=1}^{4} K_j$ and $m \geq 2$ is the length of the largest variadic AC argument list occurring in t. For convenience we will assume logarithms are base two. The basis cases are trivial. We consider the two induction cases. When $t = f(t_1, \ldots, t_n)$ we have

$$\begin{aligned} cost(t) &\leq K_3 n + \sum_{i=1}^{n} cost(t_i) \\ &\leq K_3 n + (K \log m) \sum_{i=1}^{n} |t_i| \log |t_i| \quad \text{(by induction hypothesis)} \end{aligned}$$

Now

$$\begin{aligned} & cost(t) \leq (K \log m)|t| \log |t| \\ \Leftrightarrow \quad & K_3 n + (K \log m) \sum_{i=1}^{n} |t_i| \log |t_i| \leq (K \log m)|t| \log |t| \\ \Leftrightarrow \quad & (K \log m) \left(\log |t| + \sum_{i=1}^{n} |t_i| (\log |t| - \log |t_i|) \right) \geq K_3 n \\ \Leftarrow \quad & (K \log m) \sum_{i=1}^{n} |t_i| \log \frac{|t|}{|t_i|} \geq K_3 (n-1) \end{aligned}$$

Now for all t_i except for possibly the largest we have $|t|/|t_i| \geq 2$ and $K \log m > K_3$ so the above chain of inequalities holds. When $t = F^*(t_1, \ldots, t_n)$ we have

$$\begin{aligned} cost(t) &\leq K_4 \left(\sum_{i=1}^{n} |t_i| - \max_{i=1}^{n} |t_i| \right) + \sum_{i=1}^{n} cost(t_i) \\ &\leq K_4 \left(\sum_{i=1}^{n} |t_i| - \max_{i=1}^{n} |t_i| \right) + (K \log m) \sum_{i=1}^{n} |t_i| \log |t_i| \quad \text{(by induction hypothesis)} \end{aligned}$$

Now

$$cost(t) \leq (K \log m)|t| \log |t|$$

$$\Leftrightarrow \quad K_4 \left(\sum_{i=1}^{n} |t_i| - \max_{i=1}^{n} |t_i| \right) + (K \log m) \sum_{i=1}^{n} |t_i| \log |t_i| \leq (K \log m) |t| \log |t|$$

$$\Leftrightarrow \quad (K \log m) \left(\log |t| + \sum_{i=1}^{n} |t_i| (\log |t| - \log |t_i|) \right) \geq K_4 \left(\sum_{i=1}^{n} |t_i| - \max_{i=1}^{n} |t_i| \right)$$

$$\Leftarrow \quad (K \log m) \sum_{i=1}^{n} |t_i| \log \frac{|t|}{|t_i|} \geq K_4 \left(\sum_{i=1}^{n} |t_i| - \max_{i=1}^{n} |t_i| \right)$$

Now for all t_i except for possibly the largest we have $|t|/|t_i| \geq 2$ and $K \log m > K_4$ so the above chain of inequalities holds. Finally, since $m < |t|$ we have $cost(t) = O(|t| \log^2 |t|)$. □

Gramlich and Denzinger [14] give a combined flattening and (merge) sorting algorithm which is superficially attractive since it requires only one pass and does not create an intermediate flattened version of the term. However it has a problem in that the size of the arguments lists at each merge step depends on the grouping of the AC function symbols in the original term and in common pathological cases can result in $\Omega(|t|^2)$ running time. For example the family of terms

$$F(a_n, F(a_{n-1}, F(a_{n-2}, \ldots F(a_2, a_1) \ldots)))$$

where $a_1 \geq \cdots \geq a_n$ requires $n - 1$ merge steps and a total of

$$1 + 2 + \cdots + (n-2) + (n-1) = \frac{n^2 - n}{2}$$

term comparisons.

2.3 Basic algorithm

The basic algorithm can be divided into four steps.

1. Convert pattern and subject to ordered normal form.
2. Decompose the matching problem into a hierarchy of bipartite graphs together with ancillary data.
3. Find a compatible set of solutions to the bipartite graph matching problems and construct a system of semi-pure AC problems which encode the constraints on the remaining unbound variables.
4. Solve the system of semi-pure AC problems to get a matching substitution.

The fourth step is repeated as necessary to extract all matching substitutions from the semi-pure AC system and the third step is repeated as necessary to extract all compatible sets of solutions to the bipartite graph matching problems. The first step was discussed in the previous section; we now describe the remaining three steps. We describe each step informally and illustrate them with the following example.

$$f(F(g(a, L), g(M, b), N^2, P), G(a, g(T, a), h(Q), h(S), N, U^2), V) \leq^?_{AC}$$
$$f(F(a, b^2, c, g(a, b), g(a, c), g(b, a), g(c, b)), G(a^3, b, g(b, a), h(a), h(b)), F(a, b))$$

2.3.1 Decomposition to bipartite graphs

The decomposition of an AC matching problems into bipartite graphs labelled with smaller AC matching problems is inherently recursive and in describing our algorithm it is useful to have the concept of the level of a subterm or subproblem. The level of a subterm within a term (in ordered normal form) is defined to be the number of AC functions above it. The level of a subproblem (p', s') within the whole AC matching problem (p, s) is defined to be the level of p' within p.

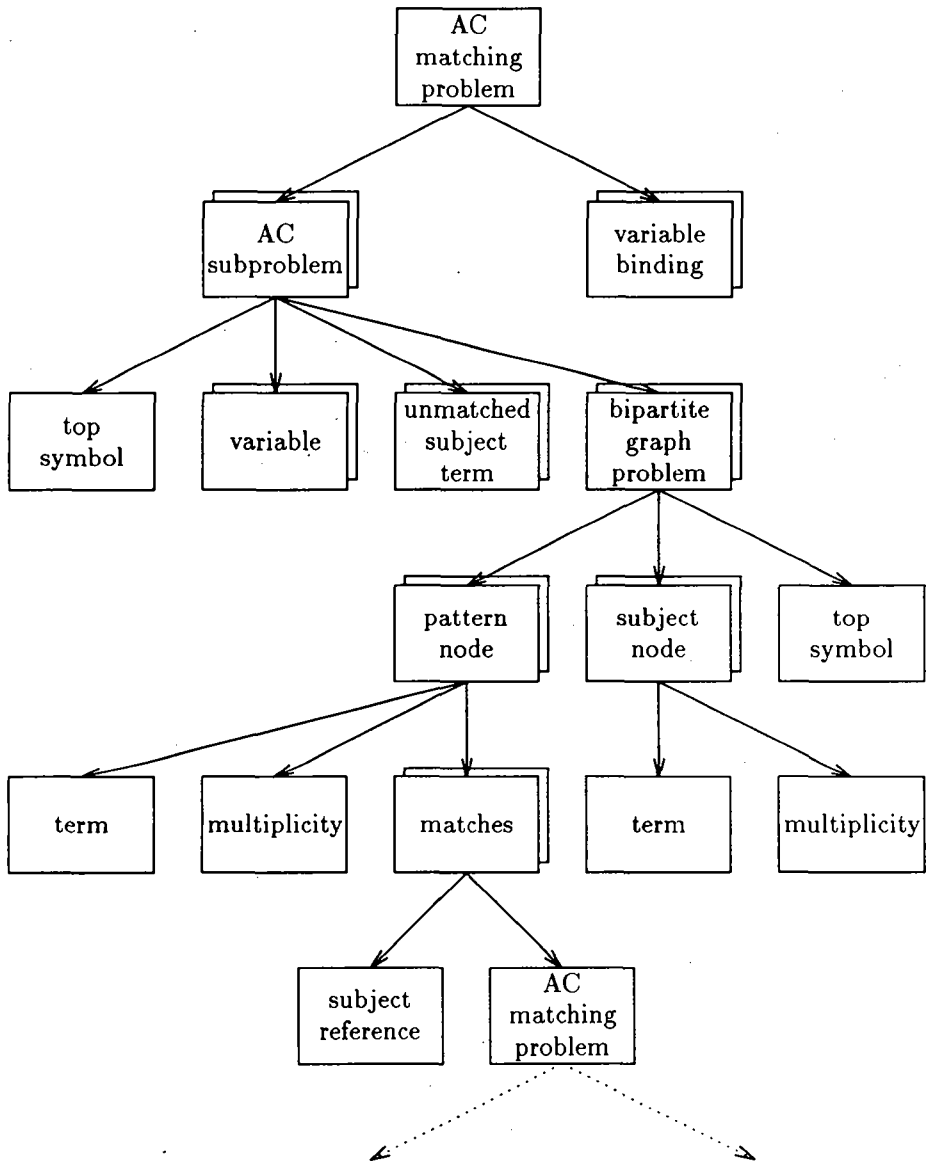

Figure 2.1: One level in the decomposition of an AC matching problem

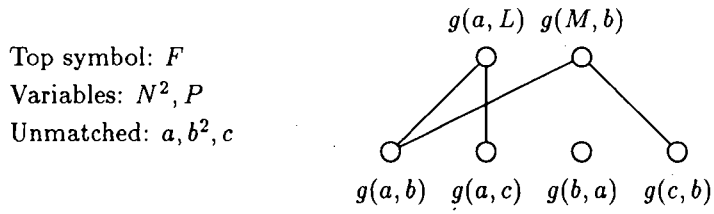

Figure 2.2: Decomposition of subproblem (2.1)

One level in the hierarchical decomposition of an AC matching problem is shown in Figure 2.1. The double rectangles indicate a (possibly empty) set of such objects, only one of which has its further decomposition explicitly shown.

If the pattern has a non-AC top symbol then there is a 'skeleton' made up of non-AC symbols in the pattern that must match those in the subject in the usual way without regard to the transpositions that can take place via the associative and commutative axioms. As a result of matching this non-AC skeleton we either have a failure or else we are left with a (possibly empty) set of variable bindings and a (possibly empty) set of subproblems whose top symbols are AC. If instead the top symbol of the pattern is an AC function symbol then we have an empty set of variable bindings and a set of AC subproblems that has the whole problem as its single member. With our example the set of variable bindings contains the single member

$$V = F(a, b)$$

while the set of subproblems with AC top symbols contains the two members

$$F(g(a, L), g(M, b), N^2, P) \leq^?_{AC} F(a, b^2, c, g(a, b), g(a, c), g(b, a), g(c, b)) \tag{2.1}$$

$$G(a, g(T, a), h(Q), h(S), N, U^2) \leq^?_{AC} G(a^3, b, g(b, a), h(a), h(b)) \tag{2.2}$$

If there is a *variable clash* (two different bindings for the same variable) we have failure.

These AC subproblems can be decomposed as follows. If there are any constant arguments to the AC top symbol in the pattern they are deleted from both the pattern and the subject if this is possible; otherwise we have failure. Now for every function symbol ϕ (AC or free) occurring directly below the AC top symbol in the pattern we form a bipartite graph where the one set of nodes (called the *pattern nodes*) correspond to the pattern subterms headed by ϕ and the other set of nodes (called the *subject nodes*) correspond to the subject subterms headed by ϕ. For each pattern/subject node pair such that the multiplicity of the subject is greater than or equal to the multiplicity of the pattern we have a subproblem of the same form as the original problem and the decomposition procedure is used recursively. If the subproblem does not decompose to failure we have an edge joining the node pair; if some pattern node has no arcs then we have failure.

Those subject terms left over (i.e. constants which are not eliminated and terms headed by functions symbols for which there is no graph) are called unmatched subject terms. If there is to be a successful match they will each have to be assigned to some variable under the AC top symbol in the pattern.

The decomposition of subproblem (2.1) is shown in Figure 2.2 and the decomposition of subproblem (2.2) is shown in Figure 2.3. The information about the hierarchy of bipartite graphs and ancillary data is stored in a data structure that closely resembles

Figure 2.1 with sets of objects being stored as pointers to linked lists or dynamically allocated arrays.

2.3.2 Solving the bipartite graph hierarchy

Bipartite graph matching problems can be solved in a naive way by selecting a match for pattern node in turn and backtracking on failure. Each time an edge is selected, any variable bindings below it are asserted and any graph problems below it are solved. Failure and backtracking are caused by variable clashes. When backtracking, before an edge is deselected, other solutions to its graph problems are

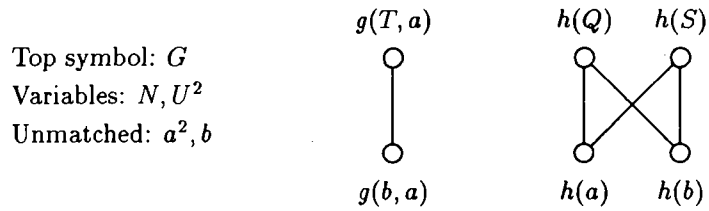

Figure 2.3: Decomposition of subproblem (2.2)

sought. When an edge is deselected any variable bindings associated with it are retracted. When all the graphs associated with an AC subproblem have a solution we are left with a semi-pure AC problem whose l.h.s. arguments consist of the variables under the AC symbol which are still unbound and whose r.h.s. arguments consist of the unmatched subterms under the AC symbol together with any subject subterms left over from the bipartite graph matching problems. The three solutions to the graph in subproblem (2.1) result in the three sets of variable bindings and semi-pure problems.

$$L = b,\ M = c,\ F(N^2, P) \leq^?_{AC} F(a, b^2, g(a,c), g(b,a))$$

$$L = c,\ M = a,\ F(N^2, P) \leq^?_{AC} F(a, b^2, g(b,a), g(c,b))$$

$$L = c,\ M = c,\ F(N^2, P) \leq^?_{AC} F(a, b^2, g(a,b), g(b,a))$$

The two solutions to the graphs in subproblem (2.2) result in the two sets of variable bindings and semi-pure problems.

$$T = b,\ Q = a,\ S = b,\ G(N, U^2) \leq^?_{AC} G(a^2, b)$$

$$T = b,\ Q = b,\ S = a,\ G(N, U^2) \leq^?_{AC} G(a^2, b)$$

2.3.3 Solving the semi-pure AC system

Suppose our semi-pure AC system consists of a single semi-pure AC problem. We can treat a pure AC problem by (notionally) replacing terms headed by function symbols under the r.h.s. AC top symbol by fresh constants. Consider a pure AC problem

$$F(X_1^{\alpha_1}, \ldots, X_n^{\alpha_n}) = F(c_1^{\beta_1}, \ldots, c_m^{\beta_m})$$

Now each variable X_i may be assigned one or more constants; if a variable is assigned more than one constant then the constants become arguments to the AC top symbol in any matching substitution. Let $x_{i,j}$ be the number of c_j's assigned to X_i. Now since under any matching substitution the multiplicity of each constant under the l.h.s. and the r.h.s. must be equal, any solution must satisfy the following system of Diophantine equations:

$$\begin{array}{ccccccc} \alpha_1 x_{1,1} & + & \ldots & + & \alpha_n x_{n,1} & = & \beta_1 \\ \vdots & & & & \vdots & & \vdots \\ \alpha_1 x_{1,m} & + & \ldots & + & \alpha_n x_{n,m} & = & \beta_m \end{array}$$

Also since every variable must be assigned something, we have the condition that for each $i \in \{1, \ldots, n\}$

$$\sum_{j=1}^{m} x_{i,j} \geq 1.$$

Each solution of the Diophantine system that satisfies this condition corresponds to a matching substitution. With a semi-pure AC system which consists of k semi-pure AC problems having the same AC top symbol we get a similar Diophantine formulation however this time we have $k.m$ equations.

Where we have a semi-pure AC system containing problems with different AC top symbols things become more complicated and r.h.s. arguments which are headed by an AC function symbol can no longer be treated as new constants. A variable that appears in two AC semi-pure problems with different AC top symbols can normally only be assigned one term from a r.h.s. argument list since if it were assigned two or more terms there would be a conflict over which AC function symbol should be used to combine them in a matching assignment. The exception occurs where one of the arguments in the r.h.s. of one semi-pure AC problem has as its head the AC top symbol of another semi-pure AC problem. For example consider the semi-pure AC system

$$F(L,M) \leq^?_{AC} F(a,b,c),\ G(L,N) \leq^?_{AC} G(F(a,b),c,d)$$

We can assign both a and b to L because when combined under the AC function symbol F we get the term $F(a,b)$ which appears in the r.h.s. argument list of the second equation; so for example

$$L = F(a,b),\ M = c,\ N = G(c,d)$$

is a matching assignment.

Our solution is as follows. First we classify each variable x occurring in the system as either *owned* by some function symbol F if x only occurs as an argument to F (though possibly as an argument to many different instances of F) or *shared* if x occurs under two or more different function symbols. If there are shared variables, a subject term $G(\alpha_1,\ldots,\alpha_n)$ under a r.h.s. AC function symbol whose head symbol G is the AC top symbol of another problem in the system must be treated specially. We check to see if $\alpha_1,\ldots,\alpha_n$ themselves occur under G as a l.h.s. AC function symbol. If so there is the possibility that $G(\alpha_1,\ldots,\alpha_n)$ might be assigned to a shared variable and we keep track of this. The actual solving procedure consists of trying to find a solution for one variable at a time and backtracking whenever failure is detected. As assignments are made, the assigned subject terms are used up and their multiplicities in the system are decremented. When making an assignment to a shared variable only one subject term may be assigned; however if it has an AC top symbol, say G, in pure AC equations headed by G, it is the arguments of G that will be checked for and 'used up' rather than the term headed by G.

Going back to our main example we have $3 \times 2 = 6$ possible pure AC systems. Taking the first solution from each graph problem we obtain a system which we write down in tabular form as

	N	P	U	a	b	$g(a,c)$	$g(b,a)$
F	2	1	0	1	2	1	1
G	1	0	2	2	1	0	0

Variable N is shared while variable P is owned by F and variable U is owned by G. Clearly the only possible assignment to N is b which leaves us with the system

	N	P	U	a	b	$g(a,c)$	$g(b,a)$
F	0	1	0	1	0	1	1
G	0	0	2	2	0	0	0

Now the only possible arguments to P and U are $F(a,g(a,c),g(b,a))$ and a. This gives us the matching substitution:

$$L = b \quad M = c \quad N = b \quad P = F(a,g(a,c),g(b,a))$$
$$Q = a \quad A = b \quad T = b \quad U = a$$

2.4 Implementation

We now discuss how the basic algorithm outlined above can be implemented. If we wish to generate all solutions it might seem natural to do a recursive search where the current state is kept on the stack frame and

each solution is returned via a call-back procedure called from the innermost recursive procedure call. There are several problems with this approach. Firstly the data structure for the hierarchy of bipartite graphs is naturally recursive so we would really like to use recursion for traversing it rather

than the search space. Secondly it is more convenient to the caller if the results are returned directly to the calling procedure so that the callers own stack frame is directly accessible. Thirdly it is useful to allow the caller to have a number of matching problems in progress at one time. This requirement arises naturally in algebraic languages such as OBJ3 [13] that allow conditional rewriting. Initially we find one solution to a matching problem and use it to do further matching and rewrites, but after the failure of a condition we wish to return to our initial matching problem and look for other solutions.

In order to handle these difficulties we will describe an implementation where recursion is only used internally to traverse data structures and to the caller the implementation appears to be nonrecursive. The caller starts by calling a procedure *build_match* with the pattern and subject as arguments. This procedure effectively implements steps 1 and 2 in the above outline and returns a *match object* which encodes the (possibly empty) set of matching substitutions. Then a second procedure *extract_match* is called with the match object as an argument when needed to extract the next matching substitution and update the match object. This second procedure effectively implements steps 3 and 4 in the above outline. Finally when either no more solutions can be found or no more solutions are wanted a third procedure *destroy_match* is called to garbage collect the match object.

This 'object oriented' approach allows many match problems to be active at once, each with a match object to hold its current state and new solutions to any particular problem can be extracted as required. The main problem with this approach is that more complicated nonrecursive search algorithms have to be used They must explicitly store their current state in the match object to allow one call to the extract procedure to backtrack over work done by the previous call to *extract_match* on the same match object.

2.4.1 Conversion to ordered normal form

Terms are represented by nodes which contain pointers to the head and tail of a singly linked list of subterms. This allows flattening to be performed in linear time using the obvious bottom up algorithm which first flattens the arguments of an AC function symbol before checking to see if either argument has the same AC function symbol as head symbol and concatenating the arguments lists if so.

The argument list of an AC function symbol in ordered normal form are stored as a sorted singly linked list of subterms and multiplicities. The conversion of a flattened term to ordered normal form is also done bottom up. The argument list of a variadic AC function symbol is first converted to ordered normal form by recursive calls. The resulting list is then merge sorted using the total ordering defined above. Finally the sorted argument list is scanned to delete duplicate terms (which will now be adjacent) and insert multiplicities. In order to simplify the management of variable bindings each unique variable symbol occurring in the pattern is associated with a unique integer between 1 and n where n is the number of distinct variables occurring in the pattern.

2.4.2 Building the graph hierarchy

The data structure representing the hierarchy of bipartite graph matching problems is constructed top-down. A recursive traversal of the pattern and subject terms is used to match any free functions, variables and constants above the top most AC function symbols(s). Variable bindings are handled as follows. An array of pointers to terms indexed by the integers associated with the variables in the pattern is maintained in the match object. This holds the current state of the variables as bindings are asserted and retracted, with unbound variables represented by null pointers.

Whenever a pattern variable is matched against a subject term we first check to see if the variable is currently bound. If it is we check the binding against the subject term; if they are not equal then we have a variable clash and we return failure for the problem.

If the variable is unbound then we enter the subterm into the array as its binding. We also enter the binding on a singly linked list of variable/terms pairs that is local to the current subproblem. This list corresponds the boxes labelled 'variable binding' in Figure 2.1. Note that by not storing bindings for variable that have already been bound (in the current subproblem or in an enclosing subproblem) in this list we are making a small optimization. During the search for compatible (with respect to variable bindings) solution to the graph hierarchy it is sufficient to bind the variable once as we can be sure that whenever the local list of bindings may be asserted the earlier binding will already have been asserted.

The AC function symbols at the current level are processed as described in §2.3.1 with variables arguments in the pattern and unused arguments in the subject being accumulated on singly linked lists. The structures for the AC subproblems themselves and the structures for the set of graph problems that they each contain are stored on doubly linked lists to enable efficient backtracking. The structures for the pattern and subject nodes of an individual graph are stored in dynamically allocated arrays; again this allows efficient backtracking and it also allows the subject nodes to be conveniently referred to by index.

The construction of the hierarchy is performed in preorder in the sense that possible matches between pattern and subject nodes in a graph are not considered until all other symbols at the top level of the current subproblem have been matched. The idea is to maximize the number of bound variables in existence when the (potentially quadratic number of) subproblems associated with graph edges are decomposed to optimize the early detection of variable clashes and hence prune the search space. The matches between pattern and subject nodes (i.e. the edges of the graph) are stored as a singly linked list of subject indices and pointers to subhierarchies for each pattern node.

When some subproblem has been completely decomposed any variable bindings that are unique to it must be retracted to avoid spurious variable clashes with sibling subproblems. This conveniently done by traversing the local list of bindings.

2.4.3 Solving the graph matching problems

The problem here is generate all sets of solutions to the bipartite graph problems that are consistent in the sense

1. A graph problem is solved iff it is at level zero in the hierarchy or it is part of a subproblem associated with a chosen edge in the solution of a higher level graph problem.

2. There is no clash between the variable bindings asserted for level zero and for each of the subproblems associated with chosen edges.

We require an algorithm such that each call to it generates the next solution or returns with a failure flag; the current state of the search being stored in the graph hierarchy data structure.

We use a set of mutually recursive procedures, each of which takes part of the graph hierarchy together with a parameter which determines whether the procedure searches for the first consistent solution to the subhierarchy (with no assumption about the current state held within the data structure) or the next solution (under the assumption that the current state held within the data structure represents a consistent solution). In the first case the procedure 'forward tracks' until it comes to a variable clash upon which it starts backtracking. In the latter case it starts backtracking until it finds a successful next solution to one of its subproblems upon which it starts 'forward tracking' Forward tracking consists of finding the first solution to each of its subproblems in turn whereas backtracking consists of trying to find a next solution to one of its previous solves problems. The basic form of a search procedure is given in Figure 2.4. In this example for simplicity we have a single recursive procedure. In practice we have three mutually recursive procedures. The first takes a problems and calls the second on each AC subproblem. The second takes an AC subproblem and calls the third on each graph. The third takes a graph and calls the first on each graph edge.

The procedures also do additional work to keep track of the solution being generated. The first asserts and retracts variable bindings; the second generates and destroys semi-pure equations while the third keeps track of which edges are selected. The semi-pure equations are pushed on and popped off an explicit stack (implemented by a singly linked list). The variable bindings are tracked using the same array as in the construction of the graph hierarchy. However it is possible that the variable may become bound to the same value several times during the solution of sibling subproblems so it is necessary to maintain a count of how many times each variable has been bound so that its binding is not retracted until all subproblems asserting it have been discarded (in favour of subproblems associated with different edges). Each time a binding is asserted the possibility of a clash is checked for and if found it leads to immediate failure of that branch of the search.

```
search(problem, reset)
    if reset then
        t = problem.first_subproblem;
    forward:
        loop
            if search(t, true) = false then
                t := t.previous_subproblem;
                goto backward
            fi;
            if t = problem.last_subproblem then
                return (true)
            fi;
            t := t.next_subproblem
        end loop
    else
        t = problem.last_subproblem;
    backward:
        loop
            if search(t, false) = true then
                t := t.next_subproblem;
                goto forward
            fi;
            if t = problem.first_subproblem then
                return (false)
            fi;
            t := t.previous_subproblem
        end loop
    fi
```

Figure 2.4: Generic procedure for searching hierarchy

2.4.4 Building the semi-pure system

In our original description of the algorithm we assumed that the semi-pure system was built during the process of solving the graph hierarchy. In a sense it is since when a consistent solution to the graph hierarchy has been found, all of the semi-pure problems will be stored in a stack. However before trying to solve the system we build a simplified array based version of it. There are several reasons for this:

1. Rearranging the semi-pure problems into an array indexed by variable numbers and term numbers greatly speeds up the solution of the semi-pure system by doing all the term comparisons at the outset. This is important since in practice the semi-pure system solving phase is executed most frequently and consumes most of the CPU time.

2. When building the array we eliminate any variables that became bound during the solution to the graph hierarchy. This can greatly reduce the search space and may lead to failure without trying to solve the semi-pure system.

3. Extra information to handle nested AC symbols correctly in the presence of shared variables is computed at this stage for efficiency.

It might seem natural to build the array based version of the semi-pure system directly and incrementally as the graph hierarchy is solved rather than push semi-pure problems on a stack. The difficulty is that we need to be able to backtrack at any point during and after the construction of a solution to the graph hierarchy. When a variable that occurs in existing semi-pure problems becomes bound we need

to eliminate it. However this also causes elimination of subject terms and even elimination of semi-pure problems themselves; which looses information and makes backtracking when we want to retract the variable binding impossible. Eliminating variables the other way around whether the variable binding is asserted before the semi-pure problem from which it is eliminated is created is fine because the semi-pure problem will be unstacked before the variable binding is retracted.

2.4.5 Solving the semi-pure system

Solving the semi-pure system is done by a non-recursive backtracking search over the possible assignments to the variables. We use three procedures, all of which take a boolean parameter that determines whether the first solution or the next solution to a piece of the problem is to be sought. The first procedure tries to find assignments to each variable in turn by calling one of the other procedures depending on whether the variable is owned or shared. On failure it backtracks and tries to find a new assignment for one of the previous variables. The procedure for finding the first or next assignment to shared variables simply tries one instance of each remaining subject term in turn. The procedure for finding the first or next assignment to owned variables is more complicated; it effectively amounts to enumerating selections from the multiset of remaining subject terms.

2.5 Refinements

We now discuss various improvements to the basic algorithm which cut down the search space, usually by early detection of paths that must fail.

2.5.1 During graph hierarchy construction

Elimination of ground terms

In an AC subproblem, ground terms occurring under the l.h.s. top symbol may be treated as constants and eliminated from both the l.h.s. and r.h.s. arguments lists (or else failure is returned if this is not possible).

Early elimination of bound variables

In an AC subproblem, if there are any bound variables under the l.h.s. top symbol then they may be substituted for by their bindings which may then be eliminated as ground terms (see above).

Subterm multiplicity sums

In an AC subproblem there are two cases to consider. If there are variables under the l.h.s. top symbol then the sum of the multiplicities of the l.h.s. subterms must be less than or equal to the sum of the multiplicities for the r.h.s. subterms. If there are no such variables then these sums must be equal. If this condition is not true then we can trivially return failure.

Node multiplicity sums

In each graph the sum of the pattern node multiplicities must be less than or equal than the sum of the subject node multiplicities or else we trivially return failure.

Elimination of trivial graphs

If a given bipartite graph problem has a single solution the subproblems associated with the solution edges must be solved and they can be decomposed at the level of the graph rather than the next level down. The graph can then be eliminated. This has two advantages. Firstly the graph does not have to be solved repeatedly (due to backtracking) during the graph solving phase. Secondly we may force variable bindings further up the graph hierarchy leading to earlier detection of variable clashes.

Elimination of unmatched subject nodes

If after all potential edges in a graph have had their subproblems decomposed some subject node has no edges then it can be deleted and its term and multiplicity are transferred to the unused term list in the current subproblem. Doing this during the graph construction phase will avoid having to do a similar operation many times (due to backtracking) during the graph solution phase.

2.5.2 During the search for hierarchy solutions

Use of polynomial time graph match enumeration algorithm

The naive backtracking method of enumerating the set of solutions to a bipartite graph matching problem can take exponential time to find a single solution. It has been known for a long time that a single solution can be found in polynomial time (the 'Hungarian method' [22]) and a more efficient algorithm is given by [15]. Recently an algorithm which enumerates the perfect matches in a bipartite graph with equal cardinality node sets using polynomial time for each solution has been discovered [9] and we can modify it to cope with our graphs in the case where all the pattern nodes have multiplicity 1.

Sorting the variables

In order to minimize the branching when searching for a solution to the semi-pure system it is desirable to choose assignmenments for the variables with fewest potential assignments first. This is done by sorting the variables so that shared variables are considered before owned variables and within those two groups variables with higher (maximum or total) multiplicity are considered before those of lower multiplicity.

2.6 Concluding remarks

We have described a new AC matching algorithm based on ordered normal forms and bipartite graph matching. We have shown that the straightforward two pass method of computing ordered normal forms is asymptotically more efficient than the one pass method of Gramlich and Denzinger. We have described a number of optimizations and implementation details which allows our algorithm to run efficiently in practice. In principle by using the polynomial time bipartite graph matching algorithm mentioned above we could obtain each solution to a linear problem instance in polynomial time which improves on the result of Benanav *et al.*

Chapter 3

Extracting Associative-Commutative Unifiers from a Diophantine Basis

Current associative-commutative unification algorithms derive one or more Diophantine equations from the problem instance, compute the Diophantine basis and generate potential unifiers by finding subsets of the basis that satisfy certain conditions. This searching process is computationally intensive. In this we examine a slightly generalized version of this searching problem and show that various subproblems are NP-complete. We derive a new search algorithm which is shown to be at worst equivalent to the best published algorithm and on average is much better. We also discuss machine level implementation techniques for this algorithm.

3.1 Introduction

The problem of associative-commutative (AC) unification is as follows: Given terms p and s we wish to find substitutions σ such that $AC \vdash p\sigma = s\sigma$ where

$$AC = \{f(X,Y) = f(Y,X), f(f(X,Y),Z) = f(X,f(Y,Z)) \mid f \in \Sigma_{AC}\}$$

is the set of associativity and commutativity axioms for some subset Σ_{AC} of the set Σ of function symbols. The classic algorithm for AC unification is due to Stickel[24]. The basic idea can be divided into four steps:

1. The left and right hand terms are flattened; that is nested occurrences of AC function symbols are replaced by a single variadic function symbol.

2. Any non-AC symbols occurring above the topmost AC function symbol are unified and discarded. Constant symbols and other nonvariable subterms occurring under the topmost AC function symbol are replaced by fresh variables; this is called *variable abstraction*. In the simplified problem each term consists of one variadic AC function symbol with variable arguments. By grouping multiple occurrences of the same variable the problem can then be represented by a Diophantine equation.

3. A basis to the set of solutions of the Diophantine equation is computed.

4. The basis is then searched for subsets of elements that yield potential unifiers. Each basis element b_j corresponds to a new variable z_j and potential unifying substitutions are formed by assigning to each old variable x_i, k instances of a new variable z_j where k is the ith component of a selected basis element b_j.

In general with multiple AC function symbols or free function symbols present the potential unifiers will themselves give rise to AC unification subproblems as the variable abstraction in step 1 is accounted for. An improved algorithm due to Kirchner[20] tries to decompose a problem instance involving free function symbols into a system of simpler problems which are solved simultaneously, reducing the search space. In this case we need to find a basis to the set of solutions of a system of Diophantine equations; however the criteria for subsets of the basis that correspond to potential unifiers is essentially the same. In solving AC unification problems, the two time consuming tasks are generating the Diophantine basis and searching it. Much effort has gone into speeding up the first of these tasks in both the single equation and system cases [16, 26, 19, 5, 4, 7, 25]. We now examine the latter task.

3.2 Searching the Diophantine basis

We will actually formulate a searching problem which is slightly more general than the problem that occurs in practice. This is because the real problem has constraints (less so in the systems case) on the basis that we cannot easily capture. Thus although we will prove that subproblems of the problem we study are NP-complete it may be that pathological cases cannot arise in practice. Nevertheless our analysis does give an insight into why the extraction of potential unifiers from a Diophantine basis is computationally expensive and allows us to formulate an improved algorithm.

Consider a Diophantine basis $S = \{s_1, \ldots, s_m\}$ associated with some AC unification problem. Each element s_j is a vector $\langle s_{1,j}, \ldots, s_{n,j} \rangle$ of non-negative integers where $s_{i,j}$ determines how many instances of a new variable z_j are to be assigned to the original variable x_i. We denote the set $\{x_1, \ldots, x_n\}$ of original variables by X. Some of the old variables, say $x_1, \ldots, x_c$ abstract constants while others, say $x_{c+1}, \ldots, x_t$ abstract terms which are neither variables nor constants; we will refer to these latter terms as alien terms. The problem is to find all subsets of S such that every original variable is assigned at least one instance of a new variable and no original variable abstracting a constant or alien terms is assigned more than one instance of a new variable. More formally we want to find all $A \subseteq S$ such that the vector

$$\langle a_1, \ldots, a_n \rangle = \sum_{s \in A} s$$

(where the summation symbol denotes pointwise summation on vectors) has no zero components and $a_i = 1$ for $i \in \{1, \ldots, t\}$.

We assume that S contains no elements which have

1. an ith component > 1 for $i \in \{1, \ldots, t\}$; or
2. ith and kth components nonzero where $i \in \{1, \ldots, c\}$ and $k \in \{1, \ldots, t\}$.

The latter condition excludes basis elements that would force the unification of a constant with another constant or alien term. We also assume that for each $i \in \{1, \ldots, n\}$ there exists some $s \in S$ with a nonzero ith component, otherwise there is trivially no solution.

In determining whether some $A \subseteq S$ satisfies the above condition it is only necessary to consider whether each component of each vector in A is zero or not. Thus we may associate each vector $s_j \in S$ with a binary vector or equivalently with the subset of original variables to which the s_j assigns a nonzero number of instances of z_j. Making use of the standard equivalence between binary vectors and subsets of finite sets, we will denote both such objects by b_j and denote the set of all b_j's by B. We will alternate between using summation notation over vectors and union/intersection notation over sets wherever it is convenient to do so. Our original problem of finding subsets $A \subseteq S$ that satisfy the above condition is equivalent to finding subsets $D \subseteq B$ that satisfy the same condition.

We consider elements in B as n-bit binary numbers with the component corresponding to x_1 as most significant, and assume that B is sorted in descending order. Thus we have a binary matrix M of the form shown in Figure 3.1.

We can divide the problem of generating a subset $D \subseteq B$ satisfying the above condition into the conjunction of three subproblems:

1. Find a subset $D_1 \subseteq \{b_1, \ldots, b_d\}$ such that $\sum_{b \in D_1} b$ has ith component 1 for $i \in \{1, \ldots, c\}$; and

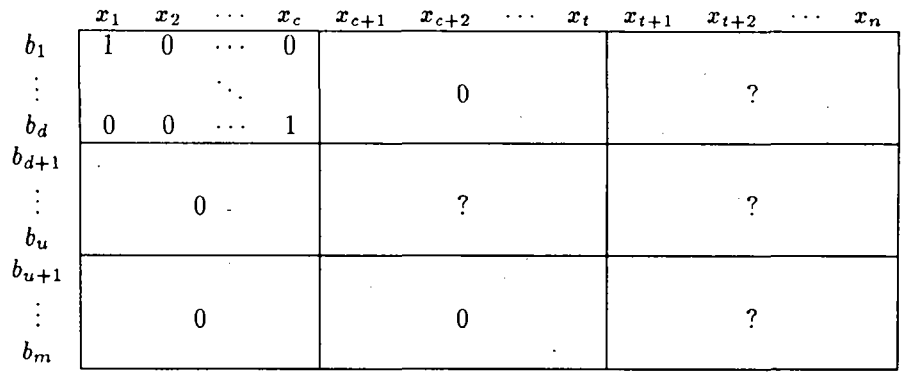

Figure 3.1: Sorted binary matrix M corresponding to Diophantine basis

2. Find a subset $D_2 \subseteq \{b_{d+1}, \ldots, b_u\}$ such that $\sum_{b \in D_2} b$ has ith component 1 for $i \in \{c+1, \ldots, t\}$; and

3. Find a subset $D_3 \subseteq \{b_{u+1}, \ldots, b_m\}$ such that $\sum_{b \in D_1} b + \sum_{b \in D_2} b + \sum_{b \in D_3} b$ has ith component greater or equal to 1 for $i \in \{t+1, \ldots, n\}$.

Notice that the solution to the third subproblem is dependent on solutions to the first two subproblems. If $\sum_{j=u+1}^{m} b_j$ has ith component nonzero for each $i \in \{t+1, \ldots, n\}$ then there will be at least one solution of the third subproblem for every combination of solutions to the first two subproblems and hence the solutions of the first two subproblems are independent. If not, there is no such guarantee and the solutions of the first two problems are interdependent and the question arises of which should be solved first.

3.3 First subproblem

We first suppose that the first subproblem is to be solved independently from the second subproblem. Now there is a single 1 in each row in the top left submatrix in Figure 3.1. For each $i \in \{1, \ldots, c\}$ let W_i be the set of b_j's where the ith component is 1 and the components $\{1, \ldots, c\} - \{i\}$ are 0. Let $|W_i| = w_i$. Then clearly $\sum_{i=1}^{c} w_i = d$ and the number of possible solutions is $w = \prod_{i=1}^{c} w_i$. Now if the first subproblem is independent, each solution can be found in at most $O(c)$ operations by a simple lexicographic enumeration algorithm. Now suppose the first problem is dependent on some solution D_2 to the second problem, i.e. $\sum_{b \in D_2} b + \sum_{j=u+1}^{m} b$ has for some values of $i \in \{t+1, \ldots, n\}$, ith components which are zero. Clearly for any solution such components cannot be zero in $\sum_{b \in D_1} b$. Thus for a brute force search we might consider all w potential solutions for D_1 to find that none of them are compatible with our solution to the second problem.

Lemma 2 $w \leq \lceil d/c \rceil^c$.

Proof: We need to bound $w = \prod_{i=1}^{c} w_i$ over all c-partitions $w_1, \ldots, w_c$ of d. Suppose we have to partition components v and v' that differ by more than one; say $v' = v + h$ for $h \geq 2$. Then we have a subproduct $v \times v' = v^2 + vh$. We can replace v and v' with $v + \lfloor h/2 \rfloor$ and $v + \lceil h/2 \rceil$ to get a larger subproduct $v^2 + vh + \lfloor h/2 \rfloor \lceil h/2 \rceil$ and hence a larger product w. Thus w is maximized when each w_i is either $\lfloor d/c \rfloor$ or $\lceil d/c \rceil$ and hence w must be bounded by $\lceil d/c \rceil^c$. □

We now show that if the first problem is dependent on the solution of the second problem then in general finding a compatible solution to the first problem becomes NP-complete. We first introduce the following decision problem which we call *cover by subset selection*: Let $S = \{e_1, \ldots, e_n\}$ and let $C_1, \ldots, C_m$ be collections of subsets of S. Does there exist for each $j \in \{1, \ldots, m\}$ some $s_i \in C_j$ such that $\bigcup_{j=1}^{m} S_j = S$?

Theorem 5 *Cover by subset selection is NP-complete.*

Proof: Membership of NP is trivial. To show NP-hardness we reduce the standard NP-complete problem *3-satisfiability* [11] to it. Consider a 3-satisfiability instance $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ where each clause ϕ_i is the disjunction of three literals drawn from a set comprising the variables $x_1, \ldots, x_m$ together with their negations $\overline{x_1}, \ldots, \overline{x_m}$. By an abuse of notation we will write $l \in \phi$ to mean that literal l occurs in clause ϕ. We construct an instance of cover by subset selection as follows. For each clause ϕ_i we have an element e_i in S. For each variable x_j we have a collection $C_j = \{S_{j,0}, S_{j,1}\}$ consisting of two subsets of S defined by

$$S_{j,0} = \{e_i \mid \overline{x_j} \in \phi_i\} \quad S_{j,1} = \{e_i \mid x_j \in \phi_i\}$$

We now show that our 3-satisfiability instance is satisfiable iff our constructed cover by subset selection instance has the answer 'yes'.

$\Longrightarrow$ Suppose our 3-satisfiability instance is satisfied by some assignment $a : \{x_1, \ldots, x_m\} \rightarrow \{0, 1\}$. Then we select subsets $S_j = S_{j,a(x_j)}$. To show that $\bigcup_{j=1}^{m} S_j = S$ for each e_i we must find some S_j that contains it. Since ϕ_i evaluates to 1 under the assignment a we know that for some x_j either $x_j \in \phi_i$ and $a(x_j) = 1$ or $\overline{x_j} \in \phi_i$ and $a(x_j) = 0$. If the first case holds then we have $e_i \in S_{j,1}$ and $S_j = S_{j,1}$ otherwise if the second case holds we have $e_i \in S_{j,0}$ and $S_j = S_{j,0}$.

$\Longleftarrow$ Suppose our cover by subset selection instance has the answer 'yes'; i.e. there exists $S_j \in C_j$ such that $\bigcup_{j=1}^{m} S_j = S$. We need to show that our original 3-satisfiability instance is satisfiable; i.e. that there exists some assignment a such that each $\phi_i = 1$ under a. We put

$$a(x_j) = \begin{cases} 0 & \text{if } S_j = S_{j,0} \\ 1 & \text{otherwise} \end{cases}$$

(Note that it is possible that $S_j = S_{j,0} = S_{j,1}$.) Now some S_j must contain e_i. If $S_j = S_{i,0}$ then $\overline{x_j} \in \phi_i$ and $\phi_i = 1$ under a. Otherwise $S_j = S_{i,1}$ and $x_j \in \phi_i$ and again $\phi_i = 1$ under a. □

Now if the first subproblem is dependent on some solution D_2 of the second problem, the first problem becomes an instance of cover by subset selection where the elements $e_1, \ldots, e_n$ are the variables whose corresponding components in $\sum_{i=u+1}^{m} b + \sum_{b \in D_2} b$ are zero and the collections $C_1, \ldots, C_m$ of subsets of S are the W_i's.

Corollary 1 *Deciding the existence of compatible solutions of the first subproblem is NP-complete.*

Proof: Any instance of cover by subset selection can be encoded as a (suitably sized) instance of the first subproblem together with an solution of the second subproblem for which we must find compatible solutions of the first subproblem. □

We chose to solve the first subproblem first and we are only interested in solutions D_1 such that

$$\left(\bigcup_{b \in D_1} b \right) \cup \left(\bigcup_{j=d+1}^{m} b_j \right) \tag{3.1}$$

since if this does not hold compatible solutions to the second and third problems will be impossible. We solve the first subproblem as follows. Clearly each $b \in \{b_1, \ldots, b_d\}$ lies in exactly one W_i and any solution D_1 will contain exactly one element from each W_i. We compute α_i for $i \in \{1, \ldots, c\}$ according to the following recursive definition.

$$\alpha_c = \bigcup_{j=d+1}^{m} b_j$$

For $i \in \{1, \ldots, c-1\}$:

$$\alpha_i = \alpha_{i+1} \cup \left(\bigcup_{b \in W_{i+1}} b \right)$$

Our nondeterministic algorithm to compute D_1 is as given in Figure 3.2. The intuition is p_{i-1} contains

$A_0 := \emptyset$; $p_0 := \emptyset$;
for $i := 1$ **to** c **do**
  choose $b \in W_i$;
  if $b \cup p_{i-1} \cup \alpha_i = X$ **then**
    $A_i := A_{i-1} \cup \{b\}$; $p_i := p_{i-1} \cup b$
  else
    fail
  fi
od;
$D_1 := A_c$

Figure 3.2: Nondeterministic algorithm to compute D_1

all the variables in X that receive non-zero assignments from the set A_{i-1} of selected vectors while α_i contains the set of all variables in X that could possibly receive assignments from the remaining vectors in $W_{i+1} \cup \cdots \cup W_c \cup \{b_{d+1}, \ldots, b_m\}$. If the test $b \cup p_{i-1} \cup \alpha_i = X$ fails then we know that by selecting b (and excluding $W_i - \{b\}$) we cannot get a solution satisfying (3.1). The nondeterminism in the 'choose' statement is handled by a straightforward nonrecursive lexicographic search where the 'fail' statement prunes the current branch by causing premature backtracking.

3.4 Second subproblem

We first suppose that the second subproblem is to be solved independently of the first subproblem.

Lemma 3 *Deciding the existence of solutions to the second subproblem is NP-complete.*

Proof: The second problem contains the standard NP-complete problem *exact cover by 3-sets* [11] as a special case and therefore must be NP-hard and it is easy to see (by nondeterministically testing all subsets of $\{b_{d+1}, \ldots, b_u\}$) that it is in NP. □

Now suppose we have a solution D_1 to the first subproblem. We are only interested in solutions to the second subproblem D_2 that satisfy

$$\left(\bigcup_{b \in D_1 \cup D_2} b \right) \cup \left(\bigcup_{i=u+1}^{m} b_i \right) = X \tag{3.2}$$

because otherwise there would not be any compatible solution to the third subproblem. For each $i \in \{c+1, \ldots, t\}$ let Q_i be the set of $b \in \{b_{d+1}, \ldots, b_u\}$ such that x_i is the least element of b (under the ordering $x_i \leq x_k \Leftrightarrow i \leq k$). Let $r \leq t - c$ be the number of non-empty sets Q_i and for $k \in \{1, \ldots, r\}$ let $\phi_k = i$ and $R_k = Q_i$ where Q_i is the kth non-empty subset.

Clearly each $b \in \{b_{d+1}, \ldots, b_u\}$ will belong to exactly one R_k and any solution to the second subproblem will contain at most one element from each R_k. We compute β_k for $k \in \{1, \ldots, r\}$ according to the following recursive definition.

$$\beta_r = \bigcup_{j=u+1}^{m} b_j$$

For $k \in \{1, \ldots, r-1\}$

$$\beta_k = \beta_{k+1} \cup \left(\bigcup_{b \in R_{k+1}} b \right)$$

Notice that β_k is independent of the solution to the first subproblem and need only be computed once for each instance of the original problem. Our nondeterministic algorithm to compute D_2 is given in

$A_0 := \emptyset$; $p_0 := \bigcup_{b \in D_1} b$;
for $k := 1$ **to** r **do**
  if $x_{\phi(k)} \in p_{k-1}$ **then**
    if $p_{k-1} \cup \beta_k = X$ **then**
      $A_k := A_{k-1}$; $p_k := p_{k-1}$
    else
      fail
    fi
  else
    choose $b \in R_k$
    if $b \cup p_{k-1} \cup \beta_k = X$ **and** $b \cap p_{k-1} \cap \{x_{c+1}, \ldots, x_t\} = \emptyset$ **then**
      $A_k := A_{k-1} \cup \{b\}$; $p_k := p_{k-1} \cup b$
    else
      fail
    fi
  fi
od;
$D_2 := A_r$

Figure 3.3: Nondeterministic algorithm to compute D_2

Figure 3.3. The intuition is that p_{k-1} contains all the variables that receive nonzero assignments from the set A_{k-1} of selected vectors while β_k is the set of all variables that could receive assignments from the remaining vectors in $R_{k+1} \cup \cdots \cup R_r \cup \{b_{u+1}, \ldots, b_m\}$. If $x_{\phi(k)} \in p_{k-1}$ then clearly no members of R_k can be included in the solution being built and unless $p_{k-1} \cup \beta_k = X$ we have failure. Otherwise we know that our solution will contain exactly one member of R_k . For $b \in R_k$ we require that b must make assignments to any variable not in $p_{k-1} \cup \beta_k$ and that b must make no assignment to variables from $\{x_{c+1}, \ldots, x_t\}$ that are in p_{k-1}. The nondeterministic choice statement is handled by a nonrecursive lexicographic search as before.

3.5 Third subproblem

We assume solutions D_1 and D_2 have been found to the first two subproblems such that (3.2) holds and give an algorithm for generating all solutions to the third subproblem such that each solution is generated in at most $O(m-u)$ set operations. We first compute γ_j for $j \in \{u+1, \ldots, m\}$ according to the following recursive definition

$$\gamma_m = \emptyset$$

For $j \in \{u+1, \ldots, m-1\}$:

$$\gamma_j = \gamma_{j+1} \cup b_{j+1}$$

Notice that γ_j is independent of the solutions to the two previous subproblems and hence need only be computed once for each instance of the original problem. Our nondeterministic algorithm to construct D_3 is given in Figure 3.4. The intuition is that p_{j-1} contains all variables in X that are given assignments by the set $D_1 \cup D_2 \cup A_{j-1}$ of selected vectors while γ_j is the set of all variables in X that are assigned to by the remaining vectors $b_{j+1}, \ldots, b_m$. The test $p_{j-1} \cup \gamma_j = X$ determines whether b_j is essential or optional to completing a solution. To generate all such D_3 the nondeterminism in the 'then' branch is handled using a stack. Note the value of p_u cannot be computed from scratch using $O(m-u)$ operations in general; however it is the union of all the b's from the solutions to the first two subproblems and is obtained as a byproduct of the generation of those solutions.

$A_u := \emptyset$; $p_u := (\bigcup_{b \in D_1 \cup D_2} b)$;
for $j := u+1$ **to** m **do**
  if $p_{j-1} \cup \gamma_j = X$ **then**
    $A_j := A_{j-1}$; $p_j := p_{j-1}$ **or** $A_j := A_{j-1} \cup \{b_j\}$; $p_j := p_{j-1} \cup b_j$
  else
    $A_j := A_{j-1} \cup \{b_j\}$; $p_j := p_{j-1} \cup b_j$
  fi
od;
$D_3 := A_m$

Figure 3.4: Nondeterministic algorithm for computing D_3

3.6 Implementation techniques

We represent each $b \in B$ by a bit vector. Each bit vector is broken into three parts, constant (components $1, \ldots, c$), alien term (components $c+1, \ldots, t$) and variable (components $t+1, \ldots, n$). For the sake of machine efficiency each part is restricted to a maximum of 32 components so that it will fit in a machine word.

In the preprocessing step we sort B to give the matrix M and find the top and middle portions. These are then divided up into the W_i's and R_k's which are held as arrays of pairs of indices (first element, last element) into the array holding M. The values of the α_i, β_k and γ_j in the above algorithms are also stored as bit vectors and are computed in reverse order; that is from γ_m though to α_1. Notice that the total number of bit vector operations to compute all these values is bounded by $O(m)$. Also for the α_i and γ_j we only need the variable part and for γ_m we only need the alien term and variable parts.

The main algorithm consists of nesting the three search algorithms given above. The current (partial) solution is stored in an array of fixed size m and which also serves as the stack; backtracking always consists of trying to remove/change the last element added to the partial solution.

Set unions and intersections are computed with bitwise 'or' and 'and' operations in the usual way. The set membership operation in the algorithm for the second subproblem in handled as a set intersection with a (precomputed) singleton set. The comparison of the result of a set operation to the set X is done by comparing the appropriate parts of the result with precomputed bit vector masks. In the algorithms for the first and third subproblems only the variable part need be considered and in the algorithm for the second subproblem only the alien term and variable parts need be considered.

3.7 Comparison with Hullot-Boudet method

Hullot[17] gives an elegant method for searching a powerset $\mathcal{P}(S)$ for subsets s that satisfy a pair of relations 'big enough' (denoted $>(s)$) and 'small enough' (denoted $<(s)$) where these relations obey the monotonicity conditions:

$$>(s) \wedge s \subseteq s' \Rightarrow >(s')$$
$$<(s) \wedge s' \subseteq s \Rightarrow <(s')$$

Intuitively if s is 'big enough' then any subset of S that contains all the members of s will also be 'big enough' while if s is 'small enough' then any subset of s will also be 'small enough'. This algorithm depends solely on these relations and takes no advantage of any structure in the elements of S.

Hullot's method can be viewed as recursively partitioning a lattice of subsets which are spanned by a pair of binomial trees. This is shown in Figure 3.5 for the set $S = \{e_1, \ldots, e_m\}$ for the case where $m = 4$. For conciseness and to better illustrate the structure of problem, subsets are labelled by binary vectors in the obvious way so that for example 0101 denotes $\{e_2, e_4\}$. We start with the assumption that $<(\emptyset)$ holds (always true for the intended application and can be checked easily otherwise). Thus we have a lattice of potential solutions whose bottom element is 'small enough'. We test the top element 1111 to see if it is 'big enough'. If not, the entire lattice can be discarded. Otherwise we partition the lattice

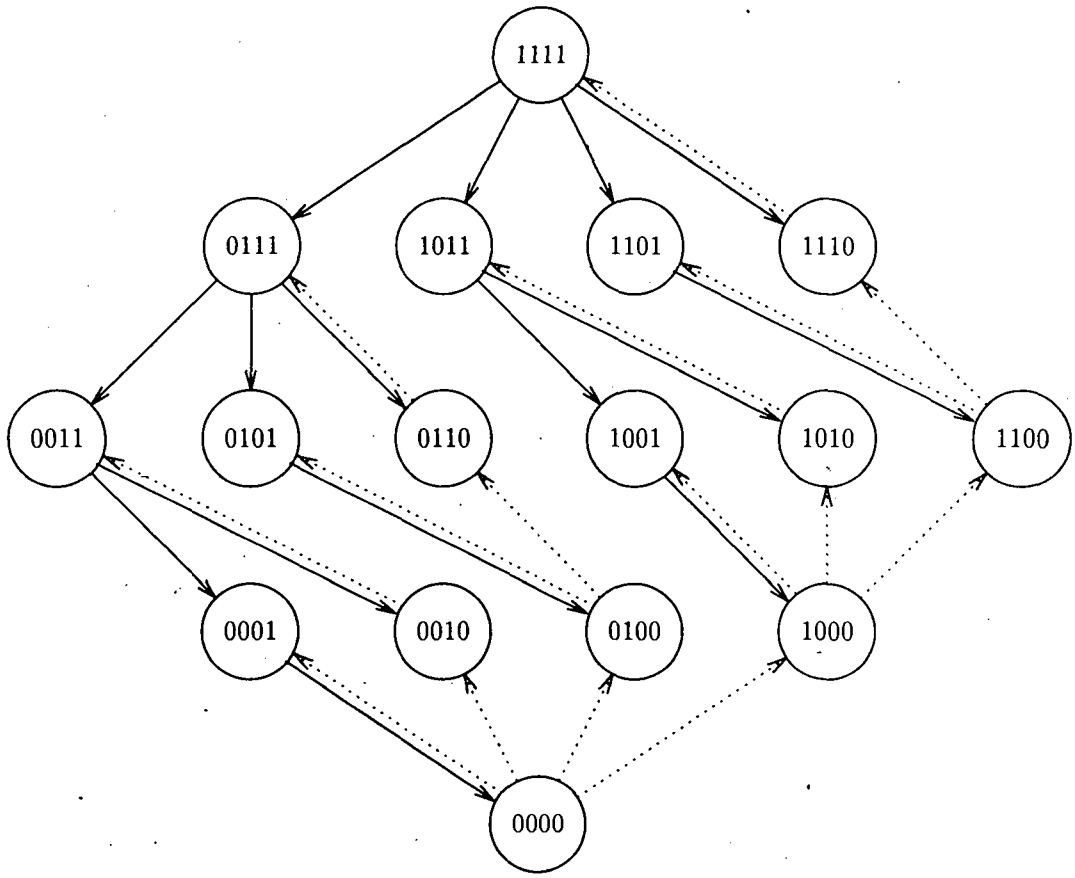

Figure 3.5: Power set spanned by binomial trees

into two sublattices, L_0, L_1 where L_0 contains all subsets which do not contain e_1 and L_1 contains all subsets which do contain e_1. Now the bottom element of L_0 is known to be 'small enough' so we test its top element 0111 to see if its 'big enough' and if not discard the entire sublattice. Symmetrically, the top element of L_1 is known to be 'small enough' so we test its bottom element to see if it is 'large enough' and if not we can discard L_2. If L_0 and/or L_1 survive this test they are themselves recursively subdivided and the process continues until we are left with single element lattices which are both 'small enough' and 'big enough'. The resulting search tree (assuming no sublattices are discarded) for $m = 4$ is illustrated in Figure 3.6. If the test at any node fails, the subtree below that node is pruned from the search.

Boudet[3] gives an efficient machine implementation of Hullot's algorithm by the restricted maximum size of the Diophantine basis to 32 and encoding each column of the (unsorted) binary matrix as a 32-bit machine word. Each potential subset s is then also represented as a machine word and the test for s 'big enough' or 'small enough' can then be done in $O(n)$ bit vector operations.

To compare our algorithm to Hullot's original algorithm we consider a simplified version which has a larger search space and show that the search space of this simpler algorithm is identical to that of Hullot's algorithm.

Assume we have an unsorted binary matrix with rows (equivalently subsets of $\{x_1, \ldots, x_n\}$) $b_1, \ldots, b_m$. We precompute the values $\delta_1, \ldots, \delta_m$ as follows

$$\delta_m = \emptyset$$

For $j \in \{1, \ldots, m-1\}$

$$\delta_j = b_{j+1} \cup \delta_{j+1}$$

If $b_1 \cup \delta_1 \neq X$ then we have immediate failure. Otherwise the nondeterministic algorithm given in

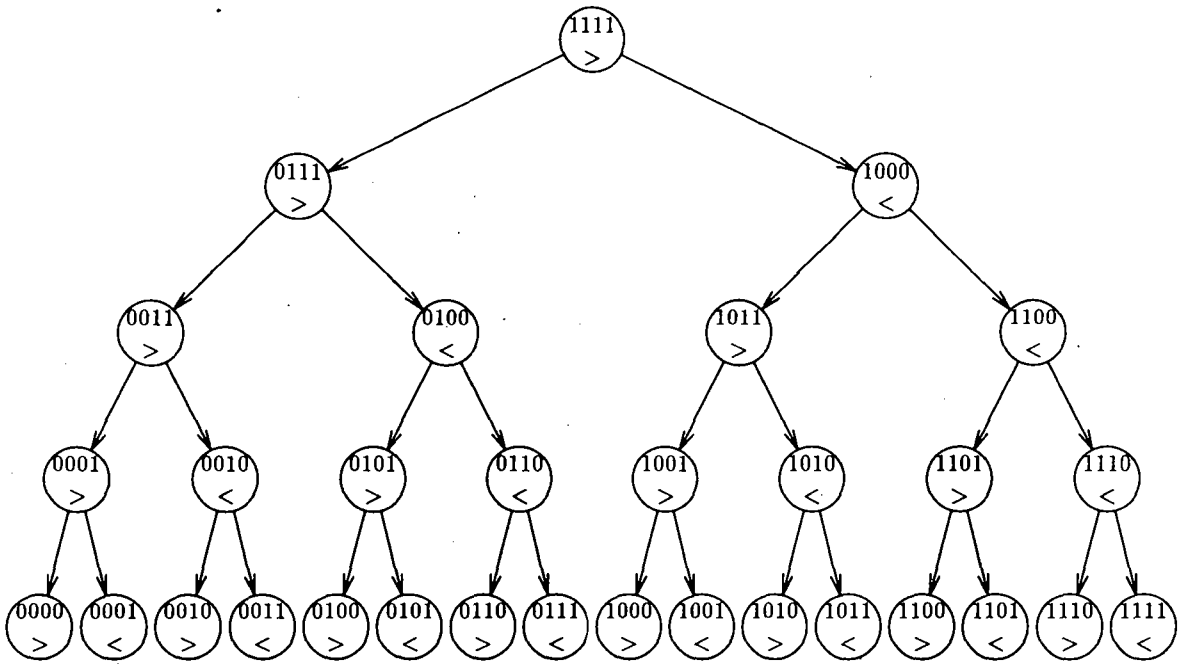

Figure 3.6: Search tree for Hullot's method

Figure 3.7 is used to search for subsets of $\{b_1, \ldots, b_m\}$. The nondeterminism is handled with a stack as before. The full search tree generated by this method for $m = 4$ is shown in Figure 3.8. Here A_j holds the set of choices already made about which elements from $\{b_1, \ldots, b_{j-1}\}$ will appear in the solution.

The test $b_j \cap p_j \cap \{x_1, \ldots, x_t\} = \emptyset$ determines whether $A_j \cup \{b_j\}$ is small enough while the test $p_j \cup \delta_j = \{x_1, \ldots, x_m\}$ determines whether $A_j \cup \{b_{j+1}, \ldots, b_m\}$ is large enough. Failure of the first test causes the right subtree to be pruned while failure of the second test causes the left subtree to be pruned. Thus at each step we consider the bottom element of the sublattice formed by including b_j:

$$\{A_j \cup \{b_j\} \cup T \mid T \subseteq \{b_{j+1}, \ldots, b_m\}\}$$

and the top element of the sublattice formed by excluding b_j:

$$\{A_j \cup T \mid T \subseteq \{b_{j+1}, \ldots, b_m\}\}$$

The search space of the simplified algorithm is thus identical to that of Hullot's algorithm.

However our main algorithm divides the problem into three subproblems and makes use of mutually exclusive sets of basis elements in the first two subproblems to cut down the search space. Also by solving the first two problems first such that D_1 satisfies (3.1) and D_2 satisfies (3.2) we detect failure in a branch before considering the third subproblem (which can always be solved).

From a machine implementation point of view our approach has two other advantages over Boudet's approach. Firstly, by using the precomputed α's, β's and γ's we require $O(1)$ n-bit operations for each test as opposed to $O(n)$ m-bit operations.

Secondly, at machine level, our approach of encoding each row of the matrix as three machine words places a limit of 32 on each part of a basis element for machine efficiency which is less restrictive than limiting the size of the basis to 32 since typically $|S| > |X|$.

3.8 Concluding remarks

We have examined a computationally intensive searching problem that arises in current AC unification algorithms. For a slightly generalized version of the problem we can show that it contains two

$A_1 := \emptyset$; $p_1 := \emptyset$;
for $j := 1$ **to** m **do**
  if $b_j \cap p_j \cap \{x_1, \ldots, x_t\} = \emptyset$ **then**
    if $p_j \cup \delta_j = X$ **then**
      $A_{j+1} := A_j \cup \{b_j\}$; $p_{j+1} := p_j \cup b_j$ **or** $A_{j+1} := A_j$; $p_{j+1} := p_j$
    else
      $A_{j+1} := A_j$; $p_{j+1} := p_j$
    fi
  else
    if $p_j \cup \delta_j = X$ **then**
      $A_{j+1} := A_j \cup \{b_j\}$; $p_{j+1} := p_j \cup b_j$
    else
      fail
    fi
  fi
od

Figure 3.7: Simplified algorithm

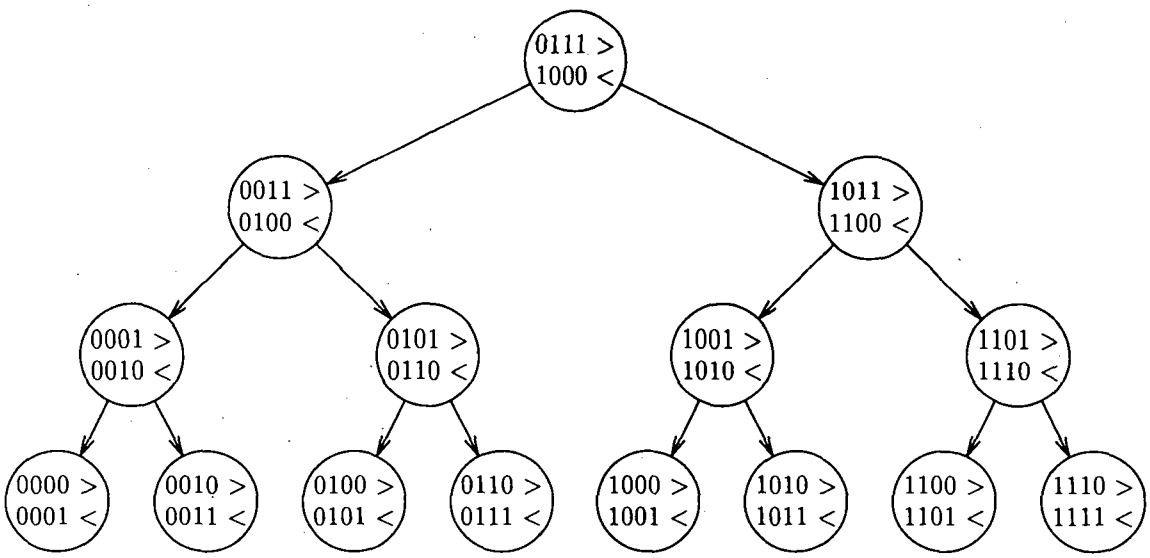

Figure 3.8: Search tree for 'simplified' method

NP-complete subproblems. Since our generalization consists of ignoring conditions on the problem instances that are difficult to capture it is hard to see how improved algorithms could be devised for the non-generalized problem that would not also apply to the generalized problem. We have derived a new algorithm for the searching problem which never explores more of the search space than the best algorithm in the literature and usually explores far less. We have discussed how the new algorithm can be implemented in practice.

Acknowledgements

I thank Claude Kirchner and Eric Domenjoud for useful discussions on the material in this report and David Duce, Chris Reade and Brian Mathews for their comments on earlier drafts. This work was done during a Fellowship funded by the European Consortium for Informatics and Mathematics (ERCIM). Most of the work was done at INRIA Lorraine with final revisions being made at Rutherford Appleton Laboratory.

Bibliography

[1] M. Adi and C. Kirchner. AC-unification race: The system solving approach, implementation and benchmarks. *Journal of Symbolic Computation*, 14(1):51–70, 1992.

[2] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3:203–216, 1987.

[3] Alexandre Boudet. Competing for the AC-unification race. Technical report, LRI, Université Paris-Sud, Bât 490, 91405 Orsay, France, 1991.

[4] Alexandre Boudet, Evelyne Contejean, and Hervé Devie. A new AC unification algorithm with an algorithm for solving diophantine equations. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 289–299. IEEE Computer Society Press, 1990.

[5] M. Clausen and A. Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8:201–216, 1989.

[6] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[7] Eric Domenjoud. Solving systems of linear Diophantine equations: An algebraic approach. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, pages 141–150. Springer-Verlag, 1991.

[8] Eric Domenjoud. AC unification through order-sorted AC1 unification. *Journal of Symbolic Computation*, 14(2):537–556, 1992.

[9] Komei Fukuda and Tomomi Matsui. Finding all the perfect matchings in bipartite graphs. Technical Report B-225, Department of Information Sciences, Tokyo Institute of Technology, Oh-okayama, Meguro-ku, Tokyo 152, Japan, 1989.

[10] R. M. Gallimore, D. Coleman, and V. Stavridou. UMIST OBJ: a language for executable program specifications. *Computer Journal*, 32(5):413–421, 1989.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[12] S. J. Garland and J. V. Guttag. An Overview of LP, the Larch Prover. In *Proceedings of 3rd International Conference on Rewriting Techniques and Applications*, number 355 in Lecture Notes in Computer Science, pages 137–151. Springer Verlag, 1989.

[13] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ3. In J. A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ*. Cambridge University Press, (to appear).

[14] Bernhard Gramlich and Jörg Denzinger. Efficient AC-matching using constraint propogation. Technical report, FB Informatik, Universität Kaiserslautern, Pf. 3049, D-6750, Kaiserslautern, Germany, 1988.

[15] J. E. Hopcroft and R. M. Karp. A $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal of Computing*, 2:225–231, 1973.

[16] G. Huet. An algorithm to generate the basis of solutions to homogenous linear diophantine equations. *Information Processing Letters*, 7(3):144–147, 1978.

[17] J.-M. Hullot. Associative commutative pattern matching. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 406–412, 1979.

[18] J.-M. Hullot. *Compilation de Formes Canoniques dans les Theories Equationnelles.* PhD thesis, University de Paris Sud, Orsay, France, 1980.

[19] Deepak Kapur. An improved upper bound for nonnegative basis solutions of a linear diophantine equation. Draft, Department of Computer Science, State University of New York at Albany.

[20] Claude Kirchner. From unification in combination of equational theories to a new AC-unification algorithm. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 171–210. Academic Press, New York, 1989.

[21] E. Kounalis and D. Lugiez. Compilation of pattern matching with associative-commutative func tions. In A. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT '91*, Lecture Notes in Computer Science 493, pages 57–73. Springer-Verlag, 1991.

[22] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:225–231, 1955.

[23] Patrick Lincoln and Jim Christian. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, 8:217–240, 1989.

[24] Mark E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, 1981.

[25] A. P. Tomas and M. Filgueiras. A new method for solving linear constraints on the natural numbers. In P. Barahona, L. M. Pereira, and A. Porto, editors, *EPIA 91 (5th Portuguese Conference on Artificial Intelligence)*, pages 30–44. Springer-Verlag, 1991.

[26] H. Zhang. An efficient algorithm for simple diophantine equations. Technical Report 87-26, Department of Computer Science, RPI, Troy, NY, 1987.

Unité de Recherche INRIA Lorraine
Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)

Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399